FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Key Management in Partitioning Operating Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Jürgen Broder

Matrikelnummer 0425784

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Peter Puschner
Mitwirkung: Proj. Ass. Dipl.-Ing. Dr. Armin Wasicek

Wien, 13. Januar 2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Jürgen Broder
Alsegger Straße 38/11, 1180 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

Embedded systems play an increasingly important role in our daily lives. They often handle sensitive data and interact with the real environment, where a malfunction or failure of the system software can cause considerable damage. Apart from the requirement to make (software) systems failure-resistant (*safety*), the protection of such systems from malicious users and software is playing an increasingly important role (*security).* This work focuses on the security aspect in embedded systems and starts out with a general introduction to this topic. An essential component of security in software systems are isolation techniques. They can be used to partition components in a resource-sharing system according to certain criteria and through this control the execution of programs. Examples of isolation techniques are sandboxing, virtualization, hardware-based isolation and isolation enforced by the operating system kernel. They provide the theoretical foundation for the further study of this work.

The guiding example for a security-related application used in this work is a cryptographic key-management program in which the stored keys represent highly sensitive data worthy to be protected. Therefore in the first step, a key management program is implemented on Linux, because it is often used as an operating system in the embedded domain. Based on this implementation, we show that Linux in a basic configuration can be considered as not secure because it does not enforce strict isolation between applications and thereby allows the exploration of the keys in the management program. In contrast, *Partitioning Operating Systems* implement isolation techniques and other security mechanisms by design. Porting the key management application to such an operating system shows that a program isolation can be strictly enforced, thus keys can be stored securely.

Kurzfassung

Eingebettete Systeme spielen eine immer wichtigere Rolle in unserem täglichen Leben. Oftmals verarbeiten sie sensible Daten und interagieren mit der realen Umwelt, wodurch Störungen oder Ausfälle der Systemsoftware erheblichen Schaden bewirken können. Neben der Anforderung (Software-) Systeme ausfallsicher zu machen (*Safety*), nimmt der Schutz solcher Systeme vor böswilligen Benutzern und Schadsoftware eine immer bedeutendere Rolle ein (*Security*). Diese Arbeit konzentriert sich auf den Sicherheitsaspekt in eingebetteten Systemen und beginnt mit einem allgemeinen Einblick in das Thema Security. Ein wesentlicher Bestandteil für die Sicherheit in Software-Systemen sind *Isolationstechniken*. Sie können dazu verwendet werden um Komponenten in einem resource-sharing System nach gewissen Kriterien zu partitionieren und so die Ausführung von Programmen zu kontrollieren. Beispiele für Isolationstechniken sind Sandboxing, Virtualisierung, Hardware-basierte Isolation und Isolation durch den Betriebssystem-Kernel. Sie liefern den theoretischen Unterbau für die weiteren Untersuchungen dieser Arbeit.

Ein klassisches Beispiel für eine sicherheitsrelevante Anwendung ist ein kryptographische Schlüsselverwaltungsprogramm in dem gespeicherte Schlüssel hoch sensible Daten darstellen die es wert sind geschützt zu werden. Daher wird im ersten Schritt ein Schlüsselverwaltungsprogramm auf Linux implementiert, da es als Betriebssystem für eingebettete Systeme häufig zum Einsatz kommt. Anhand dieser Implementierung wird gezeigt, dass Linux in einer Basiskonfiguration als nicht sicher eingestuft werden kann, da es keine strikte Isolation zwischen Programmen erzwingt und dadurch das Auslesen der Schlüssel im Verwaltungsprogramm erlaubt. *Partitioning Operating Systems* hingegen implementieren Isolationstechniken und andere Sicherheitsmechanismen per Design. Anhand einer Portierung der Schlüsselverwaltungsapplikation auf solch ein Betriebssystem zeigt sich, dass eine Programmisolation erzwungen werden kann und somit Schlüssel sicher gespeichert werden können.

Contents

List of Figures

List of Abbreviations

ABI	Application Binary Interface
ACL	Access Control Lists
API	Application Programming Interface
ASIC	Application-specific Integrated Circuit
ASIP	Application-specific Instruction Set Processor
ASLR	Address Space Layout Randomization
CISC	Complex Instruction Set Computer
DAC	Discretionary Access Control
DMA	Direct Memory Access
DRM	Digital Rights Management
FPGA	Field Programmable Gate Array
GDB	GNU Debugger
IPC	Inter-process Communication
ISA	Instruction Set Architecture
KDC	Key Distribution Center
KDT	Key Distribution Tool
LKM	Loadable Kernel Module

LOC	Lines of Code
MAC	Mandatory Access Control
MCP	Maximum Controlled Priority
MILS	Multiple Independent Layers of Security
MLS	Multilevel Security
MMU	Memory Management Unit
NGSCB	Next Generation Secure Computing Base
NIST	National Institute of Standards and Technology
PCC	Port Communication Channel
PCP	Partition Communication Port
PID	Process Identifier
POSIX	Portable Operating System Interface
PSSW	PikeOS System Software
RBAC	Role-based Access Control
RISC	Reduced Instruction Set Computer
RNG	Random Number Generator
RPC	Remote Procedure Call
RTE	Run Time Environment
RVM	Reference Validation Mechanism
SKDP	Secure Key Distribution Protocol
ST	Security Target
TCB	Trusted Computing Base
TCG	Trusted Computing Group

TCPA	Trusted Platform Alliance
TLB	Translation Lookaside Buffer
TPM	Trusted Platform Module
VM	Virtual Machine
VMIT	Virtual Machine Initialization Table

CHAPTER 1

Introduction

1.1 Motivation

Embedded Systems play an important role in today's world and can be found in almost every modern electronic system like personal computers, notebooks, network devices, mobile phones, domestic appliances, consumer electronics, smart cards, and more sophisticated applications like sensor networks in cars, trains, airplanes and even space shuttles. Those systems often need to access, store, communicate, and manipulate sensitive data and probably need to distribute this data over a network, making security a major issue to their design [RRKH04]. Contrary to general purpose computer systems, embedded systems do massively interact with the physical world. In systems where safety is a serious issue, breaking the security could lead to property damage, personal injury, and even death [Koo04]. A constraint of many embedded systems is that they often have to operate within limited resources and physically insecure environments. Therefore security in embedded systems is not just an additional feature like implementing cryptographic algorithms and protocols, but more a whole design process in which other metrics like cost, performance and power have to be taken into account [KLMR04].

1.2 Problem Statement

A classic example of a security-related application is a cryptographic key-management program in which the stored keys represent highly sensitive data worth to be protected. One possible utilization of such a key-managemenent software could be that applications request for keys in order to encrypt data they want to send to other applications locally or even over some network interface. Another example would be a "*Secure Group Communication Service*" where applications share a cryptographic key as a common secret. Then all applications in posession of the shared secret are allowed to participate in the group communication. In both cases there will also be applications that are not allowed to know about those keys, which forms the major security requirement for the key-management software. Through this *separation* we call applications *trusted* if they are, and *untrusted* if they are not allowed to request for cryptographic keys.

An essential component of security in software systems are isolation techniques. They can be used to partition components in a resource-sharing system according to certain criteria and through this control the execution of programs. Examples of isolation techniques are sandboxing, virtualization, hardware-based isolation and isolation enforced by the operating system kernel. Hence they can also be used to separate the key-management application from other applications to securely store the cryptographic keys.

Linux is used in many embedded systems. It enforces separation by the virtual memory management model only, which might not be enough for protecting sensitive data in applications. Additionally it controls access to resources like system memory over a user model that may also not be applicable for the embedded domain. Compared to that, a *Partitioning Operating System* implements separation (isolation) and security features by design. Summarizing , the goal of this work is to compare the architectural features of the monoltihic Linux operating system to those of the PikeOS partitioning operating system. This will be done by implementing the key-management software on both systems and then examine the differences and probable benefits of each, including some penetration testing techniques.

1.3 Chapter Overview

Chapter 1 gives an introduction to this thesis. *Chapter 2* will be concerned with general security issues and security topics in the embedded systems domain. In *Chapter 3* we will work out popular isolation techniques, including sandboxing, virtualization, operating system kernels, and hardware-based isolation. This chapter forms the theoretical foundation for the further examinations. *Chapter 4* gives an implementation overview of the examplary key-management software. *Chapter 5* forms the biggest part of this work. It gives a short architectural overview of both, the Linux and PikeOS operating systems. After defining a use case it defines some memory acquisition cases based on Linux, that show how cryptographic keys can be explored from the key-management software. Afterwards those cases will be compared to the implementation on PikeOS. For each case there is a discussion that lines out the benefits and differences between both operating systems. *Chapter 6* concludes this thesis.

CHAPTER 2

Basic Concepts and Related Work

2.1 Security in Embedded Systems

2.1.1 Overview

This chapter gives an overview of the topic of computer security. In the first phase of the work on this thesis, this survey was done in order to get an understanding of basic ideas and concepts concerning secure computer systems. In [UP95] the term computer security is defined as:

> *"The protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability, and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications)."*

This already includes the three primary security objectives behind information and computing services. The security of any computer system can be reasoned about by examining their presence or absence. They are defined as follows in the literature [SB08], [BB04], [Sta08]:

- **Confidentiality**: Data confidentiality means that sensitive data may not be disclosed to unauthorized users or applications. Data privacy assures that individuals are able to decide which information related to them is collected and stored and to whom that information may be disclosed.

- **Integrity**: Data integrity ensures that information and programs may only be changed by authorized users or systems. System integrity on the other hand means that the system needs to perform its intended function.
- **Availability**: States that the system should work promptly. No unauthorized persons or systems should be able to deny service access to authorized users.

Besides those three primary objectives, there are several secondary objectives which are strongly coupled to the topic of computer and operating system security. In some sense they can bee seen as combinations of some of the primary objectives:

- **Authentication**: In order to distinguish between legitimate and illegitimate users of system services and for ensuring integrity, availability and confidentiality of a system, we need mechanisms that allow users or systems to be identified [OK09]. This means that we need to verify that users are who they claim to be, and that their input to the system comes from a *trusted* source. Notice here that in [BB04] the term authenticity is included under integrity.
- **Authorization:** An authorization mechanism manages the access of authenticated users to system resources. This often is done by enforcing a *security policy* which associates users with certain rights or permissions.
- **Accountability:** Since truly secure systems are not yet achievable, it is necessary to uniquely trace down sources of security breaches. Therefore systems must keep records of their activities so that they are able to trace security breaches or revert security critical actions.

When designing and implementing computer systems, engineers have to take into account these basic objectives to build a system with a certain degree of security and thus protecting it against a defined set of malicious attacks. Today there is an increasing number of embedded systems which often need to compute sensitive information and distribute it over the Internet or other kinds of public networks. Therefore the (financial) loss due to insecurity, can be very high. With the rapid development of new technologies the attack surface is expanding, making it a duty to deal with security. Moreover, embedded systems often operate in a constrained environment, making security in embedded systems a hardware-software design challenge. In [RRKH04] those constraints are described as the *Processing Gap, Battery Gap, Flexibility, Tamper Resistance, Assurance Gap* and *Cost*. The processing gap and battery gap means that embedded systems are often limited in their processing capabilities and have to deal with limited battery capacities. They also need to be flexible to the rapid development of new technologies and therefore need to adapt to new security requirements. The term tamper resistant means that embedded systems must not only be secure against software attacks, but also against physical and side-channel attacks. *Assurance* is concerend with building reliable systems, which in general is a difficult task. Therefore it is clear that securing such systems is also quite challenging. Last the cost factor is very influential on the security architecture for an embedded system, since many of them need to be manufactured in high quantity.

The design process of a secure embedded system starts with a definition of requirements under consideration of the constraints named before. In addition it is essential to know about the threats against which a system should be protected. In the following chapters we will introduce basic cryptographic algorithms and some security solutions built upon them. We will also have a short look at common threats to embedded systems.

2.1.2 Security Terms

The key discipline of computer security is the protection of *system resources*, also called *assets*. System resources can be prone to three general categories of *vulnerabilities*. They may be *corrupted*, can become *leaky*, or may become *unavailable*. These three classes can be linked to the primary security objectives of *integrity, confidentiality* and *availability*, mentioned before. Corresponding to the various types of vulnerabilities, there are different *threats* that are able to exploit those vulnerabilities, whereby a threat is a potential security breach to a system resource. In further consequence an *attack* is a threat actually being carried out. If the attack was succesful, the security of the system will be compromised. The instance carrying out an attack is called the *attacker* or *threat agent*. In order to deal with such security attacks, various *countermeasures* are introduced to the system, which primarily try to prevent those attacks. If an attack could not be prevented, countermeasures have to deal with attack detection and recovery. It is important here to say that the implementation of countermeasures can introduce new vulnerabilities to the system.

2.1.3 Computer Security Strategy

So where do we have to start in order to introduce security to our embedded systems? Lampson describes three aspects that need to be involved in a comprehensive security strategy [Lam04]:

- Specification / Policy
- Implementation / Mechanisms
- Correctness / Assurance

2.1.3.1 Security Policy

The basic needs for information security are described by the security objectives mentioned before and aim at protecting assets from attackers. So the first step before we can design security services and mechanisms is to describe the needs for security in our system. This is done by specifying a *security policy*. It usually describes who should or should not have access to a certain resource and is at least an informal description of the system behavior. The security policy quotes the requirements for reaching the *primary objectives* of confidentiality, integrity and availability (see chapter 2.1.1). The security policy must be enforced by technical controls of the system.

2.1.3.2 Security Implementation

The aim of a security implementation is to defend the system against vulnerabilities which can originate from bad programs or agents. This can be done by the well-known *access control* model (see figure 2.1), in which a *guard* also called a *reference monitor* enforces the security policy by regulating the requests for service to valued resources. Those resources are usually encapsulated in *objects*, whereby the sources of requests are called *subjects*. The guard uses *authentication* and *authorization* mechanisms to decide if the subject is allowed to carry out a certain operation on the object. On the other hand the *information flow control* model is somehow dual to the access control model (see figure 2.2). It is sometimes used if we want to preserve secrecy in the presence of bad programs. Here the guard decides if information can flow to a principal. For both of the models stated before, Lampson defines authentication, authorization and accountability as the gold standard for implementing security (see chapter 2.1.1).

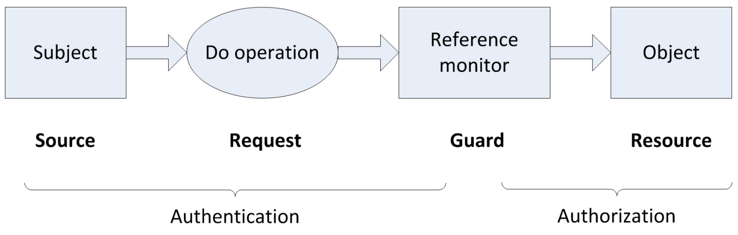


Figure 2.1: Access Control Model

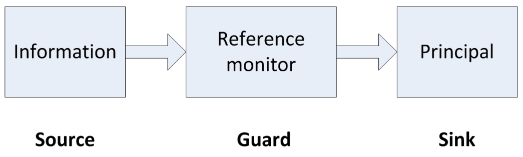


Figure 2.2: Information Flow Control Model

2.1.3.3 Assurance and Evaluation

So when security mechanisms have been implemented we somehow need to check that they work as intended. This can be achieved by the two concepts of *assurance* and *evaluation*. The NIST Computer security Handbook defines [GR95]:

> *"Computer security assurance is the degree of confidence one has that the security measures, both technical and operational, work as intended to protect the system and the information it processes."*

Therefore we can say that assurance deals with the question if our system meets its requirements and suitably enforces its security policy. Assurance only states a degree of confidence and must not be confused with a formal proof.

Evaluation on the other hand is about examining our system with respect to certain criteria. This may include testing and formal or mathematical techniques. The main goal of the work done in this area is to develop evaluation criteria which then can be applied to any security system. The *Common Criteria IEC 15408 Standard* [Cri12] for example defines criteria for evaluating different products.

2.1.4 The Concept of Trusted Systems

2.1.4.1 Security Models

We already heard that maintaining security in embedded systems is a design and implementation issue and can cause certain problems. It is difficult to design hardware or software modules which *assure* that the design provides an intended level of security. This difficulties in design may lead to many unanticipated security vulnerabilities. But even if we can show that our design is correct, it is almost impossible to implement the design without errors, which is another source of possible vulnerabilities. Therefore we need a method to *prove* either logically or mathematically, if our design satisfies the defined security requirements and if the implementation meets the design specification. This need for a prove led to the development of formal computer security models like the *Bell-LaPadula* [BL73] [BL76], *Biba* [Bib77], *Clark-Wilson* [CW87], and *Chinese Wall* [BN89] models, that can be used to verify security designs and implementations. These models are referred to as *multilevel security (MLS)* models. They enforce access control of applications to system resources based on different security levels, which can also be seen as different privilege levels. Users or applications get assigned to one of the security levels. Information flow between applications is then determined by the security levels. On every access different rules will be applied corresponding to the security model. For more details check out the references.

2.1.4.2 Trusted Computing Base

The security models described before aim at enhancing the *trust* that users and administrators have in the security of a system. The concept of *trusted systems* goes back to the early 1970s. The aim was to first develop such security models and then to implement hardware and software platforms to achieve trust in the system. The central approach to trusted systems is the concept of a *reference monitor*, which was described in the Anderson Report [And72]. We introduced the concept of a reference monitor in chapter 2.1.3.2. A reference monitor is used for access control decisions between subjects and objects. Anderson describes the set of hardware, software and setup information (the firmware) implementing the reference monitor as the *Reference Validation Mechanism (RVM)*. It is responsible to enforce the system's security policy and must have the following properties [SB08]:

- **Complete Mediation:** the security policy must be enforced on *every* access to the system resources.
- **Isolation:** This means that the reference monitor has to be protected against unauthorized modification. An attacker must not be able to change the logic of the reference monitor.
- **Verifiability:** The correctness of the reference monitor must be provable. This means to show mathematically that it enforces the security policy and provides complete mediation and isolation. If this is the case, the system is said to be *trustworthy*.

After the development of some prototypes it was clear that the concept of a *Reference Validation Mechanism (RVM)* alone is insufficient, because there are other functionalities in the system, like administrating or auditing functions, that can affect the correct operation of the RVM. In 1981, Rushby described an implementation of a RVM as a *security kernel* [Rus81]. The primary motivation in kernelized systems is to isolate and localize all security critical software in the kernel. Then the security models (e.g. *Bell-LaPadula*) are used to verify if the kernel is secure. If this is the case, all non-kernel software becomes irrelevant to the security of the system. Soon it turned out that many applications are different from that what was stated in multilevel models. The attempts to implement such applications on conventional kernels with a system-wide security policy have led to systems with considerable complexity, where the verification turned out to be a very difficult task. Rushby describes by an example why there was the need to introduce *trusted processes* outside the kernel, which are able to violate the rules of multilevel models. With the introduction of trusted processes, the kernel is no longer the sole arbiter of security. It has to be guaranteed that the special privileges that are granted to the trusted processes are not abused by those processes and not usurped by other, untrusted processes. The combination of the kernel and trusted processes is called the *Trusted Computing Base (TCB)* of the system [AJ95]. So in order to guarantee the security of the system we need to verify the entire TCB.

Trusted systems are a central concept to security. Much development in this area is based on the works of Rushby [Rus81] and [Rus84], which primarily deal with the *separation* of *trusted* components from *untrusted* ones. Components can either be hardware or software. In this work we will focus on software components. In order to securely execute software, untrusted programs then can be rejected or *sandboxed* and thus be isolated from trusted ones. We will revert to this in chapter 3. With his work, Rushby laid the foundation for the *Multiple Independent Levels of Security* architecture, that we will deal with when evaluating key management in our Partitioning Operating System (see chapter 5.7.1). The *Trusted Computing Group* is a non-profit standardization organization also dealing with the enforcement of trust in systems. We will have a short look at their *Trusted Computing Technology* and the *Trusted Platform Module (TPM)* in chapter 2.2.

2.1.4.3 Terminology

To sum up this chapter we will list some terms for trusted systems, taken from [SB08]:

- **Trust:** The extent to which someone who relies on a system can have confidence that the system meets its specifications (i.e., that the system does what it claims to do and does not perform unwanted functions).
- **Trusted System:** A system believed to enforce a given set of attributes to a stated degree of assurance.
- **Trustworthiness:** Assurance that a system deserves to be trusted, such that the trust can be guaranteed in some convincing way, such as through formal analysis or code review.
- **Trusted Computer System:** A system that employs sufficient hardware and software assurance measures to allow its use for simultaneous processing of a range of sensitive or classified information.
- **Trusted Computing Base (TCB):** A portion of a system that enforces a particular policy. The TCB must be resistant to tampering and circumvention. The TCB should be small enough to be analyzed systematically.
- **Assurance:** A process that ensures a system is developed and operated as intended by the system's security policy.
- **Evaluation:** Assessing whether the product has the security properties claimed for it.
- **Functionality:** The security features provided by a product.

2.1.5 Threats and Attacks

2.1.5.1 Physical and Side-Channel Attacks

Physical and side-channel attacks exploit the system implementation and/or identifying properties of the implementation. They can either be *invasive* or *non-invasive*. Invasive means that an atacker gains physical access to the appliance (opens it), in order to observe, manipulate and interfere with system internals. Examples of invasive attacks are microprobing and design reverse engineering. On the other hand non-invasive attacks do not necessarily need to open the device and gain direct access to it. Attack techniques like timing attacks, fault induction, power and electromagnetic analyses are often cheap and scalable but need a certain amount of creativity of the attacker. A system is said to be *tamper-resistant* if it is secure even if it is subject to physical and side-channel attacks [RRKH04].

2.1.5.2 Logical Attacks

The most common threat to an embedded system are logical attacks. These attacks are implemented through malicious software such as viruses, worms and trojan horses. They try to manipulate sensitive data (*integrity attacks*), disclose confidential information (*privacy attacks*), and deny access to the system resources (*availability attacks*). Due to the computational complexity, it shows that hackers do not intend to directly break cryptographic primitives employed in security mechanisms. Instead, most software attacks will target vulnerabilities in the implementation of functional security mechanisms and their cryptographic algorithms. This vulnerabilities arise from bugs in the operating system or downloaded application code [RRKH04]. In [KLMR04], Kocher describes three factors as the source of vulnerabilities in embedded systems software:

- **Complexity:** Software in modern systems is complex and will even become more complex in the future. It is clear that with larger software also bugs and security vulnerabilities will appear more often. The problem gets even worse with the usage of unsafe programming languages like *C* and *C++*, which do not supply protection mechanisms against simple attacks like buffer overflows. But due to their simplicity this programming languages are very popular for embedded systems. In theory we would need to proof that the software is error-free which is in fact impossible for the complex and large software in todays systems.

- **Extensibility:** Modern embedded systems are designed to be extensible. To provide end users a richer functionality and increased customizability, the embedded system needs to be able to download and execute *untrusted* software. It's obvious that such software can introduce new vulnerabilities to the system's security.
- **Connectivity:** Today it is quite common that embedded systems are connected to the Internet, which makes them even more vulnerable to attacks since hackers don't need to gain physical access anymore. A small failure in a software could propagate through the network and cause some severe security breaches.

In addition, Ravi describes some common design and implemention problems which are the source of vulnerabilities including: buffer overflows, failure to secure code update process, use of insecure cryptographic algorithms, cryptographic protocol flaws, key management failures, random number generator defects, use of debug modes that bypass security, improper error handling, incorrect algorithm implementation, security parameter negotiation weaknesses, sequence counter overflows, improper reuse of keys, poor user interfaces, use of weak passwords, operator errors, pointer errors, OS weaknesses, solving the wrong problem, inabilitiy to reestablish security after compromises, and so on [RRKH04]. To address those issues it is necessary to develop appropriate hardware and software countermeasures. We will have a look at some general approaches in the following chapter.

2.1.6 Security Mechanisms

2.1.6.1 Countermeasures

As already stated before, logical attacks try to exploit design or implementation flaws in the software of the system. We heard about the need for flexibility and execution of downloadable code in embedded systems. The issue with downloadable code is that it cannot be trusted and can be malicious, so running it directly on the host could cause severe damage to the system. But also the operating system code could contain bugs that make systems vulnerable. Therefore the aim of security design is to develop countermeasures that ensure the confidentiality and integrity of sensitive code and data during every stage of execution and to determine with certainty that it is safe from a security standpoint to execute a given program.

The first issue can be addressed through the introduction of different ***hardware*** and ***software techniques***, which regulate the access of software components (operating system, downloaded code, etc.) to various portions (registers, memory regions, security co-processors, etc.) of the system [RRKH04], [RRC04].

- usage of dedicated hardware to protect memory locations, e.g.: *Discretix Cryptocell* [Dis11]
- secure bootstrapping [AFS97]
- usage of cryptographic file systems [Bla93] [GSMB03]

On the other hand, ***secure software execution*** can be achieved through:

- software authentication and validation checks
- restricted environments for code execution and isolation of programs through *sandboxing* and *virtualization* (see chapter 3)
- detection of security policy violations through *run-time monitors* [KBA02]
- usage of safety proof carrying code [Nec97]

Effective countermeasures must *guarantee* a certain degree of security in the system from the powered-on state. Therefore most security measures define notions of *trust* or *trust boundaries* across the different hardware and software resources (see chapter 2.1.4). This enables the system to detect violations, like for example illegal access to memory regions, between trust boundaries. So trust boundaries are a natural and convenient basis for the system to decide about the security or the breach of security by itself. As we have already mentioned, this work will focus on design and implementation issues for secure software execution. More on this in chapters 3 and 5.

2.1.6.2 Cryptographic Ciphers

Cryptographic algorithms form the theoretical basis of information security and can be used as building blocks to implement security mechanisms in order to reach certain objectives. In further consequence they achieve encryption, decryption and checking the integrity of data. We distinguish between *symmetric*, *asymmetric* and *hashing* ciphers:

- **Symmetric Ciphers:** Establish only one secret key for encryption and decryption of data that is sent from a sender to a receiver. Therefore they primarily provide *data confidentiality*. Some symmetric cipher examples are *DES*, *3DES*, *AES*, *IDEA*.
- **Hashing:** They convert messages into unique fixed length-values which provides signatures and integrity checks on messages. Examples: *MD5*, *SHA*.
- **Asymmetric Ciphers:** Also known as *public-key encryption*. They use a private (secret) key for decryption and a related public (nonsecret) key for encryption and verification. With the usage of the private key, asymmetric ciphers can provide user or host *authentication*. They are typically used in security protocols like *IPSec* and *SSL*. In combination with hashing they can be used to construct *digital certificates*. Examples: *RSA*, *Diffie-Hellman*.

2.1.6.3 Security Solutions

Most functional security mechanisms like security protocols are designed and implemented around cryptographic ciphers to achieve security objectives. Most of these mechanisms need keys in order to work. Therefore a good and secure *key management* (see chapter 4) is central to almost every security solution. Examples for various security solution are listed in the following [RRKH04]:

- ***Secure Communication Protocols*** like *IPSec* and *SSL* are used to ensure secure communication channels from and to the embedded system.
- ***Digital Certificates*** are used to identify entities.
- ***Digital Rights Management (DRM)*** like *OpenIPMP*, *MPEG*, *ISMA*, and *MOSES* are used for application protection against unauthorized usage.
- ***Secure Storage*** and ***Secure Execution*** is used to tailor the system architecture for security considerations.

But these solutions alone cannot ensure security. We heard that embedded systems are often resource-constrained, where designers need to face the processing gap, battery gap, assurance gap, need for flexibility, tamper resistance, and cost considerations. This reveals a huge design space from which designers can choose from to implement features that provide the embedded system with required security functions. In addition those features must be implemented efficiently to come up with the different limitations.

2.1.6.4 Architectural Design Space

Kocher [KLMR04] describes the architectural design space for embedded security throughout four levels of considerations, which are illustrated in figure 2.3. The first row is about different macroarchitecture models. Designers have to decide whether they want to use embedded general-purpose processors (EP), application-specific instruction set processors (ASIP), EPs with custom hardware accelerators, and so on. This somehow can be seen as the basic, overall system architecture. The second row is about making the right choice for base processor parameters like instruction-set architectures and microarchitectures. This could be used to tune the base processors to custom needs. In the third row decisions must be made about which security processing features should be designed and how they should be implemented. Some options would be if the functionality should be implemented by custom instructions, hardware accelerators or general-purpose instruction primitives. In the last row, decisions about attack-resistant features in the embedded processor and system design are made, with the aim to protect the system against software and physical attacks. This can be be achieved by enhanced memory management, process isolation architectures, additional redundant circuitry against power analysis attaks, fault detection circuitry, and so on.

2.1.6.4.1 Security processing architectures

Basic security objectives like integrity, availability and confidentiality can be achieved through the implementation of security protocols and cryptographic algorithms. Cryptographic ciphers and protocols can be very compute intensive and power hungry. Therefore, security processing architectures are designed to support limited processing capabilities and to deal with power issues in an embedded system. There are different approaches for designing such architectures:

- **Hardware:** for example use ASICs (Application Specific Integrated Circuits) to implement whole cryptographic algorithms in hardware.
- **Software:** use only the embedded general-purpose processor (EP) core for security protocol and cryptography processing and implement security functions in software.

- **Hardware-Software:** hybrid approach, with several solutions for implementing security functions:
  - General-purpose embedded processor core with hardware accelerators.
  - Usage of FPGAs (Field Programmable Gate Arrays) to allow reconfiguration.
  - Integration of accelerator hardware with processor core, which is invoked through custom instructions, also called Application Specific Instruction Processor (ASIP).
  - Usage of new instructions in the general-purpose processor to accelerate symmetric ciphers.
  - etc.

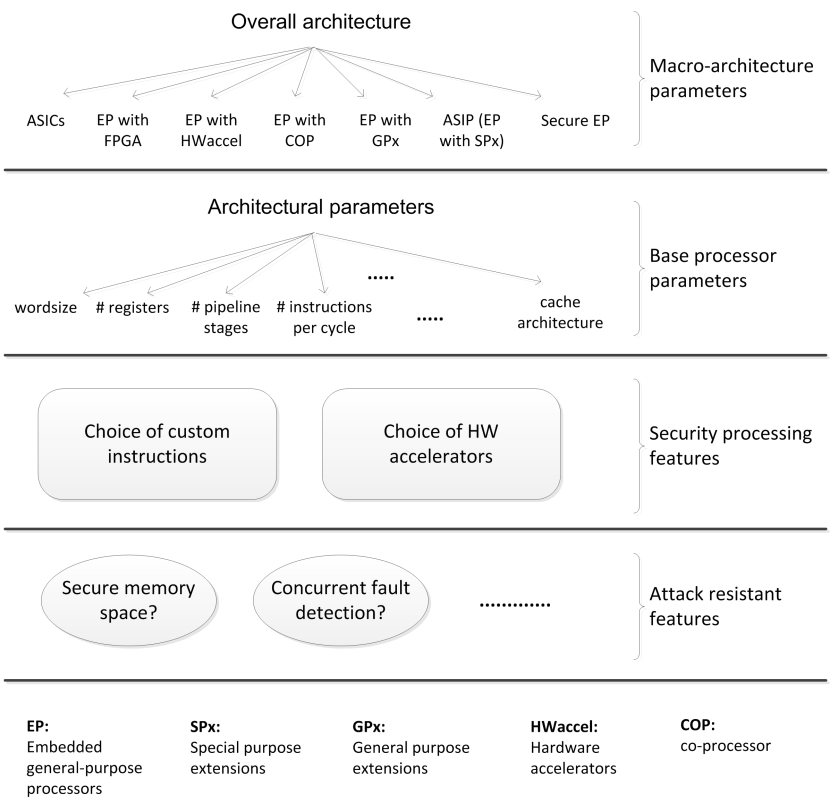

Overall architecture
ASICs
EP with FPGA
EP with HWaccel
EP with COP
EP with GPx
ASIP (EP with SPx)
Secure EP
Macro-architecture parameters
Architectural parameters
wordsize
# registers
# pipeline stages
# instructions per cycle
.....
cache architecture
Base processor parameters
Choice of custom instructions
Choice of HW accelerators
Security processing features
Secure memory space?
Concurrent fault detection?
....
Attack resistant features
EP: Embedded general-purpose processors
SPx: Special purpose extensions
GPx: General purpose extensions
HWaccel: Hardware accelerators
COP: co-processor

Figure 2.3: Architectural Design Space

2.1.6.4.2 Attack-resistant architectures

We heard before, that security processing architectures implement the basic security functions. But there is no protection against software, physical or Denial-of-Service attacks. Therefore secure embedded systems should also implement appropriate attack-resistant features.

In the industry there are some commercial initiatives that are driving the development of attack-resistant archtiectures like the *Trusted Computing Platform Alliance (TCPA)* or the *Next Generation Secure Computing Base (NGSCB)*. The work of these initiatives is based on the basic assumption, that secure and non-secure software (applications and operating systems) must be able to coexits on the system. This assumption has led to the conceptual architectural model of separate and parallel security domains, which is closely related to the concept of trust we heard before. In this model the secure domains are protected from the non-secure ones. This is often achieved by new software and hardware features that isolate secure computations and memory and protect them from corruption. The NGSCB for example defines four key objectives for secure systems:

- **Strong Process Isolation**: Protected execution of software through the introduction of a new security domain.
- **Sealed Memory:** Sensitive information is encrypted and bound to a given platform and software context. This memory can only be unsealed with a correct key in the appropriate context.
- **Platform Attestation:** Describes the abilitiy to detect changes from remote users on the system
- **Secure Path to User:** Invocation of secure programs in the trusted domain through users.

2.2 Practical Examples

2.2.1 ARM TrustZone

The TrustZone technology [Yor03] from ARM is a security extension to the ARM core architecture which was developed to provide a hardware-based protection mechanism against software attacks in embedded systems. The TrustZone approach is based on the clear separation of trusted and untrusted code. We already heard about the concept of trusted systems and separation in chapter 2.1.4. Some general approaches to *software isolation* and *sandboxing* that enforce separation will be discussed in chapter 3 and of course in chapter 5 when examining Partitioning Operating Systems and the MILS architecture.

All trusted code and applications are evolved from a *trusted code base (TCB)* that is located in a secure area of the processor. This relatively small code is responsible for regulating the security of the entire system, starting from the system boot sequence to enforcing a level of trust at each stage of a transaction. It is protected by implementing a separate secure domain that is enforced through hardware changes to the core architecture and memory system. Therefore, besides the typical separation of operating systems and applications through user and privileged modes, the system gets divided into a "normal" and "secure" domain, often referred to as "worlds" (see figure 2.4). Then trusted applications need to be identified before they gain access to the secure domain, whereas nonsecure applications are denied. The key change to the architecture and hardware, that is enforcing this access policy, is the introduction of the so called *S-bit*. This simple identifier denotes which parts of the system are secure. It is applied to the ARM core, memory system, selected peripherals, and so on. Through the S-bit the current operating state of the ARM core is defined (normal and secure). A separate processor operating mode called *Monitor* is responsible for controlling these operating states, which means controlling accesses to the S-bit. The monitor itself is accessible by a limited and predifined set of entry points only. It is also responsible to ensure a secure transition (context switch) between secure and nonsecure states, which is achieved through the manipulation of the S-bit.

One of the major benefits of the TrustZone technology implementation is the fact that it may be extended beyond the processor core to other parts of the system like the memory hierarchy and peripherals. An example implementation on an embedded system is shown

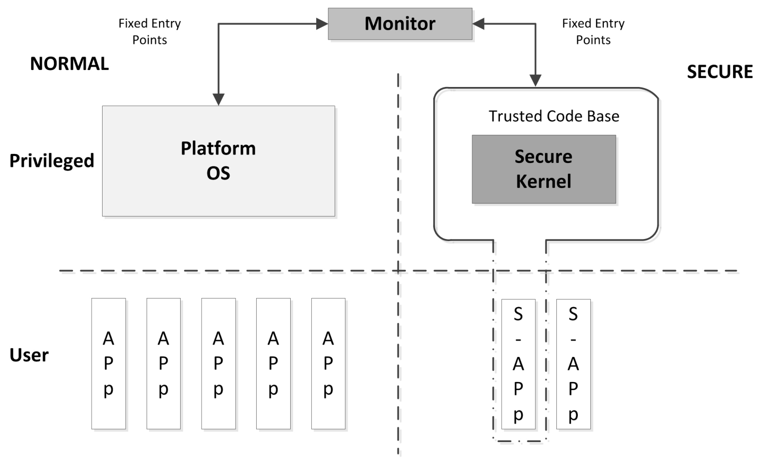


Figure 2.4: Separation of secure and nonsecure domains in ARM TrustZone [Yor03]

in figure 2.5. Here the overall system architecture also gets divided into a secure and nonsecure world. The boot code may be stored on a secure on-chip boot ROM. This is necessary since modifications to the boot process would render any security scheme ineffective. The memory also gets divided into secure and nonsecure areas. The Monitor mode in collaboration with the S-bit facility ensures that secure data won't be explored to nonsecure areas. The same division happens with exception handling. The monitor mode is used to process critical interrupts, since interrupts may freeze the processor while processing sensitive information [RRKH04].

2.2.2 Trusted Computing and the Trusted Platform Module

Trusted Computing is a standard based on *trusted systems* that was developed by the *Trusted Computing Group (TCG)*. The *Trusted Platform Module (TPM)* is an additional hardware module that forms the heart of the hardware/software approach to trusted computing. In fact the term trusted computing today is always referred to this special solution. The trusted computing technology is mainly employed in personal computer motherboards, but may also be used in smart cards, PDAs, and smartphones. Trusted computing consists of three components [Bra08]:

- **Trusted Platform:** A trusted platform is a secure computer platform that is able to store the fundamental secrets of the system like cetificates and cryptographic keys, to execute critical operations in a secure hardware environment, and to measure the

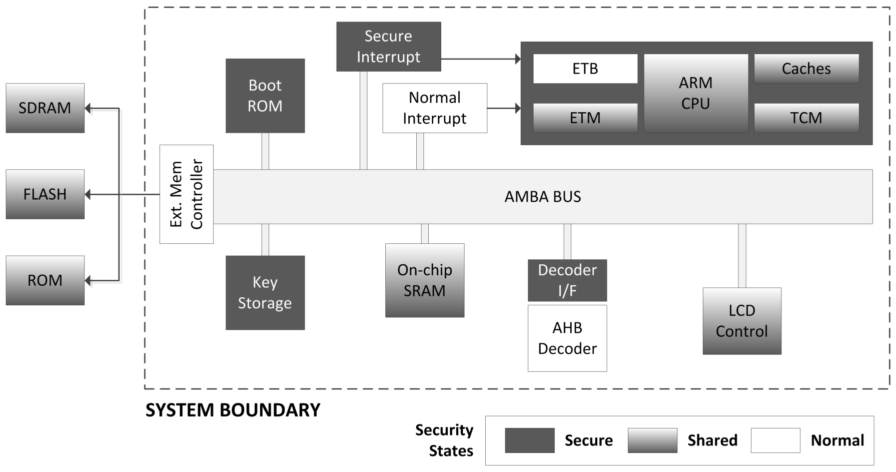


Figure 2.5: TrustZone example in an embedded system [Yor03]

integrity. Primarily this is done through the TPM. The TPM uses cryptographic mechanisms to measure the integrity of software data structures as well as the hardware and stores this values so that they can be verified. The computer's operating system, but also suitable application programs, may use those values to check if the hard- or software configuration has been changed and to react correspondingly if this is the case. The TPM is a passive module that is not able to influence or even interrupt the boot process or program sequence of the central processor. It only receives control and state measurement data from the central processor. This data then is processed, stored and read back from the TPM's safe structure. These results then are sent back to the central processor and are used to control the running of further security procedures. The TPM may also be used to securely store cryptographic keys in a hardware protected environment. The structure of a TPM with corresponding security functions are shown in figure 2.6.

- **Secure processor architecture:** Processors need additional security features in order to work with the trusted computing standard. Intel and AMD already enriched their processors with such features. In the embedded systems domain ARM also manufactures such secure processors.

- **Secure and trustworthy operating systems:** Secure operating systems are a prerequisite to use the security features of secure processors and to use the trusted platform in order to provide the user with security features. Furthermore the trusted operating system acts as the initiator of the integrity measurement and evaluation.

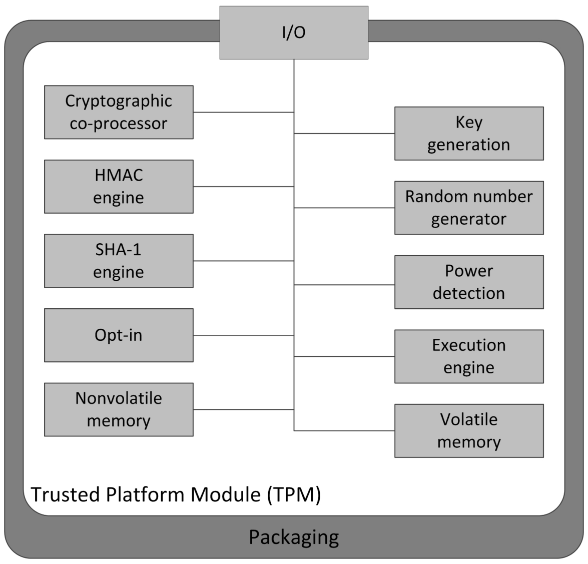


Figure 2.6: TPM Component Architecture [SB08]

Trusted computing already starts at the lowest level of the platform, beginning with the boot process. The TPM as a certified hardware security module from a trustworthy manufacturer therefore is trusted a priori. At system start a continuous security chain (*"chain of trust"*) is established from the lowest level up to the applications. As soon as a lower level disposes a stable security reference, the next level can be started upon the lower level. The TPM therefore is a hardware security reference that serves as a *"root of trust"* of the entire security chain. Already in the beginning it is verified if the signature of the platform components have changed, for example if a component has been changed or removed. Similar verification mechanisms then verify one after each other the correctness for example of the BIOS, bootblock and operating system. Therefore the system's state of security and trust is retrievable over the TPM throughout the entire boot process and also later on. Figure 2.6 shows a block diagram of the functional components of the TPM which will be described shortly in the following [SB08].

- **I/O:** The TPM components are accessed through commands over the I/O component.
- **Cryptographic co-processor:** This component implements RSA encryption/decryption, RSA-based digital signatures, and symmetric encryption.
- **Key generation:** Creates RSA public and private key pairs and symmetric keys.

- **HMAC engine:** An algorithm that is used in different authentication protocols.
- **Random number generator (RNG):** As its name implies, this component is responsible for generating random numbers that are used in a variety of cryptographic algorithms.
- **SHA-1 engine:** Implementation of the SHA algorithm used in digital signatures and the HMAC algorithm.
- **Power detection:** Power state management of the TPM and the platform power states.
- **Opt-in:** Security mechanisms to allow the TPM to be enabled or disabled.
- **Execution engine:** Here runs the program code that executes the TPM commands received from the I/O port.
- **Nonvolatile memory:** Storing persistent identity and state parameters.
- **Volatile memory:** Temporary storage. Used for execution functions, storage of current TPM state, cryptographic keys, session information and volatile parameters.

CHAPTER 3

Secure Software Execution and Isolation Techniques

3.1 Introduction

Through the growing number of network-enabled devices the number of attack mechanisms that enforce the propagation of malware in a system is also on the rise. Embedded systems need flexibility, which means that it must be possible to execute downloaded code, e.g. a firmware update. But the issue with such code is that it cannot be trusted, because it can be malicious, thus executing the code directly on the host could cause security breach to the system. On the other hand, trusted code like the operating system could also contain bugs which can make the system vulnerable. An effective and simple solution to this problems is to create *isolated* environments in order to enforce a stronger separation of functions. Then certain ill-effects in various software portions can't affect the system as a whole. Isolation techniques are a fundamental building block for the security in today's systems. The need for isolation was already described in the Anderson Report [And72], which states that the main reason for security issues in systems is the sharing of resources. Therefore the concept of the *reference monitor* was introduced, which in fact controls the execution of the programs (see chapter 2.1.3.2). Most of the isolation techniques are a variation on the generic concept of a reference monitor. Isolation of functions is established at a minimum by the operating system's kernel through *processes*, where virtual memory is abstracted from the hardware and no processes can interfere with the memory of another process. But this simple isolation has shown to be inadequate in order to deal with the various attack techniques, because many allow to access resources on other ways than via processes. In this chapter we will have a look at some other techniques that will mitigate these issues and enforce stronger isolation.

3.2 Trusted Computing Base and Protection Domains

First of all we will introduce some terms in order to define the isolation problem and introduce a generic model for isolation in systems.

- **Task:** Is an abstraction of a piece of software. It consumes resources in order to perform a specific function.
- **Shared Resource:** Can be something like a CPU, storage, or a network. Tasks perform their functions by sharing resources with other functions.
- **Protection Domain:** Is a logical container for tasks and shared resources. It enforces the protection boundary policies using isolation techniques.
- **Trusted Computing Base (TCB):** The set of software, hardware and firmware that is responsible for enforcing a security policy. The TCB can misbehave without affecting security (see also chapter 2).

With this definitions we define the task isolation problem as the problem of *seperating* and protecting tasks from other executing tasks within a protection domain and from tasks in other protection domains. Figure 3.1 shows the generic model of the isolation problem, whereby the protection domains enforce policies on the running tasks. The thickness of the protection domains states to which amount the TCB is needed to enforce the protection domain. In the model it is also possible to impelement recursive protection domains, that are protection domains within other protection domains. Figure 3.2 shows an example for a local protection domain in an operating system. Here the aim is to separate tasks running on the node from each other, and to prevent tasks from outside to access the tasks on the node [VN09].

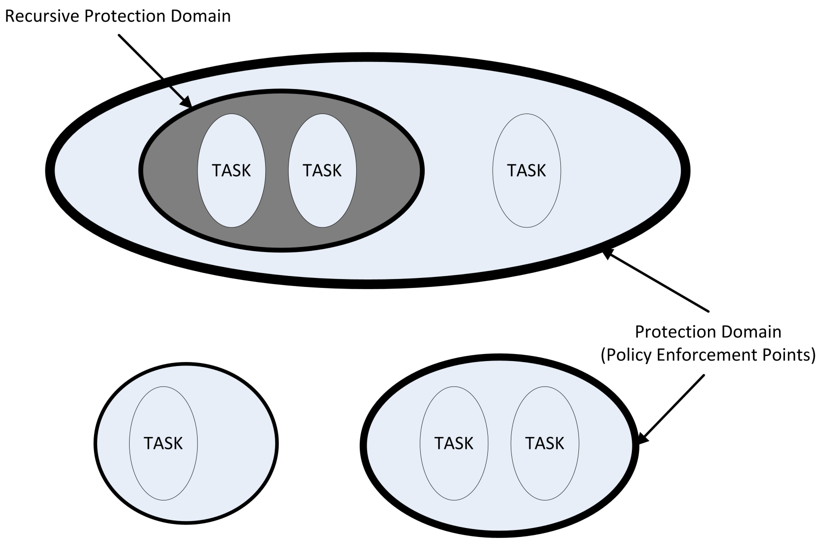


Figure 3.1: Isolation Model

3.3 Sandboxing

The first technique for separating running programs are sandboxing mechanisms. They are often used to execute untested or untrusted code which should not affect other programs. Hence they create confined execution environments to run trusted and untrusted programs on the same machine. One popular example for sandboxing is the *UNIX chroot jail*, which is used to create a *restricted virtual view* of the file system to remote users. Thereby a jail is a set of resource limits which are enforced on programs by the operating system kernel. Such resource limits may be I/O bandwidth caps, restricted filesystem namespace, disk quotas and network access restrictions. Sandboxing techniques can be devided into three coarse categories, which will be presented with examples in the following chapters.

3.3.1 Instruction Set Architecture

Instruction Set Architecture (ISA) based sandboxing techniques are all those that restrict program execution at the instruction level. One easy way to achieve restricted code execution is to rewrite the application's binary code (the object code) by inserting additional instructions around existing code like branches and stores, in order to check for memory access violations.

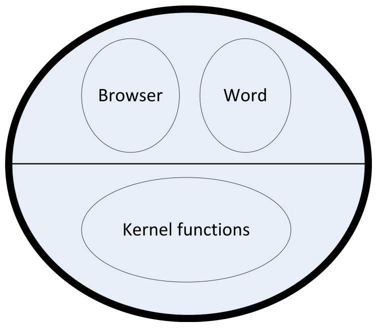


Figure 3.2: Local Protection Domain

One of the first sandboxing techniques based on binary rewriting was introduced by [WLAG93], which encapsulates software through software-based isolation. In modern operating systems, some parts can be enhanced by independently developed software modules. A very popular example for this are microkernel designs (see chapter 3.5), where such modules are implemented as user-level servers that can be easily modified or replaced. They can threaten the security of the system since they are generally distrusted and malicious software could interfere with the address spaces of other program modules. One way to prevent this would be to place every module in its own address space, where communication between address spaces is realized by *Remote Procedure Calls (RPC)*. Code in one address space is protected from code in another through hardware page tables. But transferring the control from one protection domain to the other comes with high cost. A cross-address-space RPC at least requires a trap into the operating system, copying all arguments form the caller to the callee, saving and restoring registers, switching hardware and address spaces (which on most machines means to flush the Transition Lookaside Buffer), and a trap back to the user-level. So this solution is very inefficient in systems where lots of communication takes place between programs.

Therefore Wahbe et al. [WLAG93] introduced a software-based solution which maintains fault isolation within a single address space. This approach consists of two parts. First of all every code and data for a module is placed in its own fault domain, which is a logically separate portion of the application's address space. In addition, every fault domain has its own identifier in order to access system resources like file descriptors. Second, the binary code of a distrutsted program is rewritten so that reads or writes outside its fault domain are prevented, which is called *sandboxing*. The isolated programs can communicate only via an explicit cross-fault-domain RPC interface.

The disadvantage with instruction level based sandboxing is that most of the techniques are dependent on the architecture or the type of instruction set (RISC or CISC). Other examples that fall under this category are *Program Shepherding* [KBA02], *Inline Reference Monitors* [ES00], and *PittSFIeld* [IBS02].

3.3.2 Application Binary Interface

The *Application Binary Interface (ABI)* is the low-level interface between an application or any type of program and the operating system or between an application and its libraries. ABIs define how the program code needs to be structured on machine language level. This includes calling conventions which control how the return values and parameters of functions should be handled, management of system call numbers and definition of how an application should make system calls to the operating system, data types, size, and alignment, and so on. We already heard that the sharing of resources is one of the main reasons for security issues [And72]. The key operating principle behind ISA based sandboxing is, that applications can cause very little harm if the access to resources is controlled or restricted. In modern operating systems, system calls are the only way by which applications can access system resources. So one common technique is to regulate the access by controlling system calls. This technique is often called *system call interposition*. With this technique it is possible to contain the ill-effects of any malicious code, which then won't affect other programs. There are four general approaches to create ISA based sandboxes [BCP+09] at which will have a look at in the following.

3.3.2.1 Kernel-Level Sandboxing

Kernel-level sandboxing techniques (see figure 3.3) build confined execution environments by hooking on the *system call entry table* of the operating system. All incoming system calls from user processes are redirected to a kernel-loadable module (see chapter 5.4.6). This kernel module inspects every single call. Access control is enforced by the policy engine which needs to be configured in order to allow or disallow operations on the system resources. The policy engine can either be implemented in the kernel or user space, but usually runs in user space. An explanation of user and kernel space is given in chapter 5.3.1. System calls are intercepted and handled by the hooked kernel module and are controlled in consultation with the policy engine. Innocuous system calls such as *close* and *exit* are not intercepted. They are directly executed by the kernel. Access

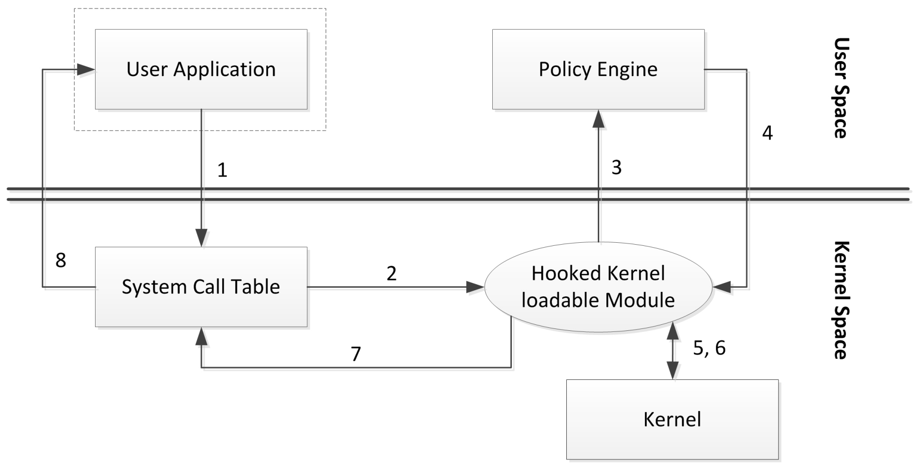


Figure 3.3: Kernel-module-based sandboxing

decisions of the policy engine are based on the user or process that is making the system call and the argument within the call. On each call the policy engine returns either allow or deny. If the call is allowed, the kernel sandbox module directly performs the current system call. The entry pointers of this call have been configured during the initialization phase. Through this method it is not necessary to change the application binary or the way how a process is invoked. The disadvantage here is that all processes have to go through this once the system call hooks have been placed. This could end up in a significant performance overhead, especially in systems with lots of system calls. *Systrace* [Pro03] and *Remus* [BGM02] are two examples for kernel-level sandboxing.

3.3.2.2 User-Level Sandboxing

In comparison to the kernel-loadable module technique, *user-level sandboxing* does not use a hook on the system call table to intercept system calls, but uses tracing features of the operating system in order to control process. One example would be the UNIX *ptrace* facility (see chapter 5.4.5), where a parent process is able to trace it's child processes. With this feature it is possible to identify the system calls invoked by a user application. It is also possible to identify the used arguments during invocation. The use of a policy engine module to decide if system calls are allowed or denied is similar to the kernel-level technique. An implementation is shown in figure 3.4.

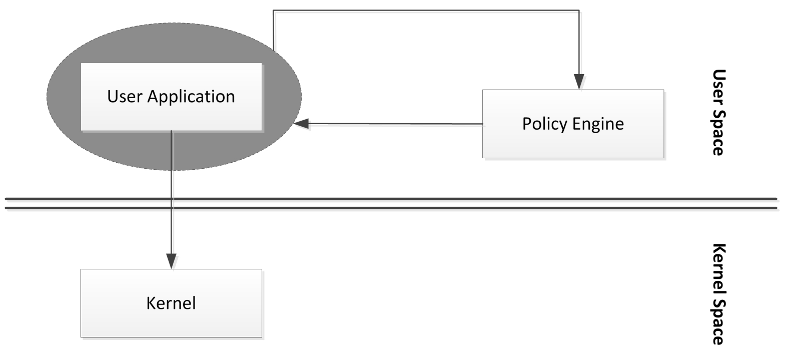


Figure 3.4: User-level sandboxing

Janus [GWTB96] is an example of user-level sandboxing technique, that implements a secure environment for untrusted applications. Here the application that needs to be sandboxed is run under a parent process. Through this it is possible to trace the permissions of the application and to trap all the system calls that are done by the application. The information about the system calls and the particular arguments of those are then sent to the policy engine. The policy engine then decides if the system call is allowed or denied. The difference to the system call interception at kernel-level is that *all* system calls will be trapped by the parent process, resulting in a higher overhead than in the kernel-based approach. In addition, the application that needs to be isolated must be run under a parent sandboxing process.

A more recent approach in modern UNIX-like operating systems is the */proc* filesystem, which is a hierarchical file-like structure that presents information about processes and other system information in a more convenient and standardized way. Usually it gets mounted to the */proc* mount point at boot time. Accessing process data held in the kernel gets more dynamically and easier than through traditional trace methods. Another example for user-level sandboxing is *Consh* [AKS99].

3.3.2.3 Delegation-Based Sandboxing

The solutions of kernel-level and user-level sandboxing techniques have the problem that the suffer from race conditions. The reason for this lies in the nonatomicity of policy evaluation and system call accesses. A so called *delegated sandboxin*g technique can mitigate such race condition problems. In this approach, after every system call invocation from a user processes, the system makes a callback to an emulation library that is located in the

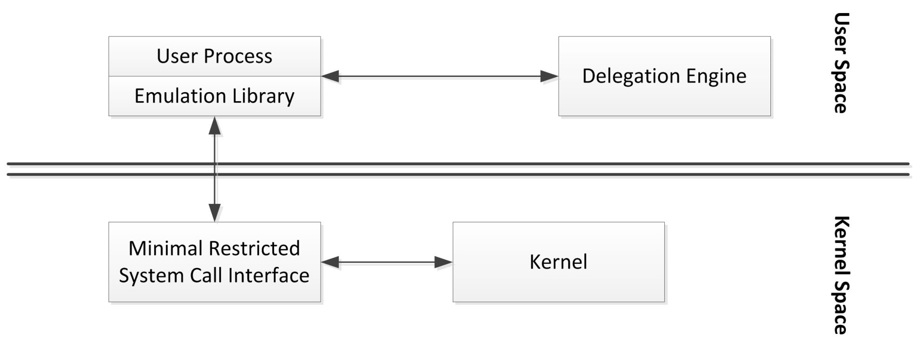


Figure 3.5: Delegation-based sandboxing

user-space. This emulation library provides the isolation service. The user-space delegation agent is called by the callback function in the emulation library. The delegation agent then executes the system call on behalf of the user application.

An example of a delegation-based sandboxing system is *Ostia* which was developed at Stanford [GPR04]. The key aspect of this system is that the delegation agent acts on behalf of the user processes in order to invoke system calls. For every user process a separate delegation agent is created. This solves the race condition and results in a minimal kernel module that can be ported easily, since the dependency on the kernel structure is minimal.

3.3.2.4 File-System Isolation

The last approach based on system call interception is called *file-system isolation*, where all file-system changes are completely isolated from the main file system. *Alcatraz* [LVS03] is an implementation of this approach which hooks into all system calls that are capable of changing the file system. Then the hook which is implemented with the system creates an isolated shadow file system. This shadow file system records all file-system changes but does not alter the primary file system. If a file needs to be updated, this approach uses copy-on-write to duplicate the file and create a separate copy. The original file will be left unchanged, whereby all child processes will use the isolated, newer version of the file. This solution can be used for complete isolation of changes to a file system by untrusted code. If the program finishes, the shadow file system will be reviewed by an administrator or another user. If they are confident that the application is trustworthy, it can be run outside the sandbox [BCP+09].

3.3.3 Access Control

We already heard about the *access control* concept in one of the earlier chapters. An access control mechanism implements a security policy that specifies which processes may have access to specific system resources. It also defines the type of access that is permitted in each instance. The access control mechanism mediates between users/processes (*subjects*) and system resources (*objects*) like files, network devices, processes, pipes, operating systems, firewalls, routers, etc. In access control mechanisms, processes are sandboxed through explicit permissions (*access rights*) that are applied to accesses by programs. Some access rights may be *read, write, execute, delete, create, search*, and so on. Access control policies dictate the types of permitted accesses, under what circumstances, and by whom they are applied.

The traditional method for implementing access control is the *Discretionary Access Control (DAC)* policy. Here accesses are based on the identity of a requestor and the access rules (authorizations) that state what the requestor is allowed to do or not to do. The policy is called *discretionary* since an entity can be granted an access right that permits the entity to enable another entity to access some resource [SB08]. The general approach of discretionary access control is usually implemented in the operating system through access control matrixes, which first were formulated by Lampson [Lam69] [Lam74]. In the matrix one dimension describes the subjects that may try to access system resources, the other dimension states the objects (resources) to be accessed (see figure 3.6 (a)). In a practical implementation the access control matrix is decomposed in two ways. The first would be decomposition by columns, yielding *Access Control Lists (ACL)*, (seen in figure 3.6 (b)). There is a seperate ACL for every object in which all users are listed with their permitted access right to the object. This is quite useful if one wants to determine which subjects have which access rights to a perticular resource. But if we want to determine which access rights are available to a specifc user, the access control matrix gets decompesed by rows, yielding *Capability Lists* (see figure 3.6 (c)). A capability ticket specifies authorized objects and operations for each user. Each user has a list of tickets and also may be allowed to give his ticket to another user. Through this, tickets may be distributed over the whole system, which makes them more security critical then access control lists. Therefore a ticket must be unforgeable. One way to achieve this would be that the operating system holds all of the tickets within a memory location inaccessible to the users. Another way would be to include an unforgeable token into the capability, which could be a large random password, or a cryptographic message authentication code.

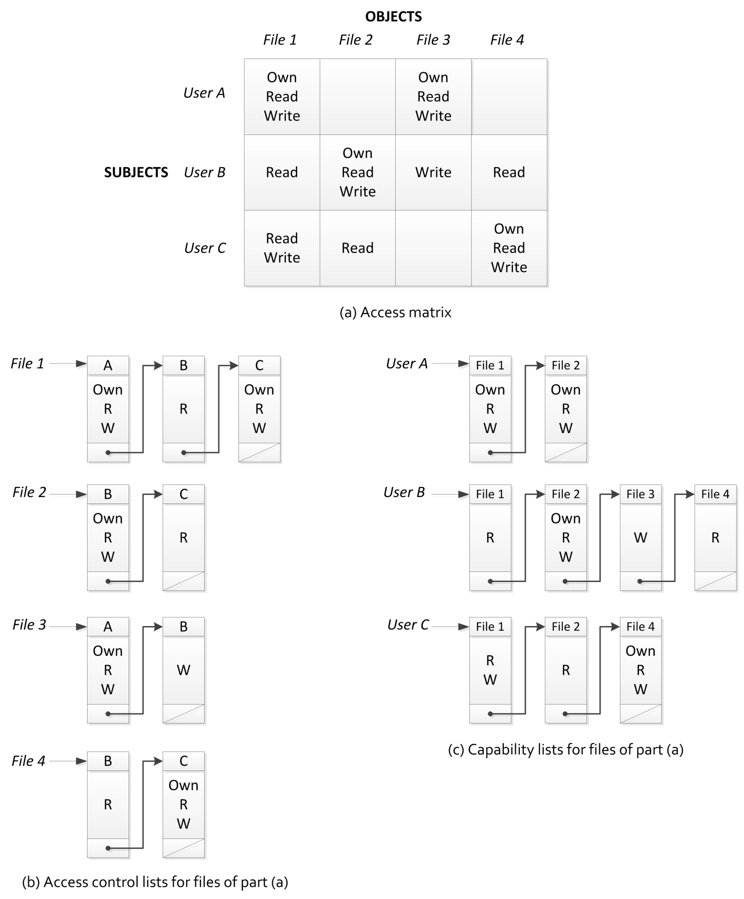


Figure 3.6: Example of Access Control Structures

The simplest example for sandboxing with access control lists is the *UNIX chroot-jail* based on UNIX's file protection mechanism. Chroot means *"change root"* and is used to switch the root directory in order to constrict the remote user's view by controlling accesses to other directories in the file system. This is a simple opportunity to quickly put untrusted software into a sandbox. But the facility has primarily not been implemented as a security feature. The problem is that a re-rooted program may not access directories and utilities outside its safe directory anymore. So those utilities need to be copied into the directory, which may cause high maintenance overhead. Therefore a lot of work has been done in order to enhance the access control approach based on the file protection mechanism. Some example systems and implementations are *TRON* [BBS95], *Sub-Operating Systems* [IBS02], and *Chakravyuha* [DMRS97].

Besides discretionary access control there also exist *Mandatory Access Control (MAC)*, and *Role-based Access Control (RBAC)* policies. Mandatory access control deals with comparing security labels of system resources and security clearances, also called security levels. We already heard about this multilevel security approach when talking about security models. This concept is often used in the context of trusted systems and we will be concerned with this in some later chapters. Role-based access control on the other hand is based on roles that users have in the system and on rules that control the accesses that are allowed to users in given roles [SB08].

As a conclusion we can say that access controlled sandboxing is more generic in its applications than ABI-based sandboxing. The ABI-based sandboxing technique relies only on preventing system calls, whereby access control sandboxing modifies the system calls in order to implement security policies [VN09].

3.4 Virtualization

3.4.1 Overview

Virtualization techniques have been around since the 1960s. In these early days they were used to logically partition mainframe computer systems into separate virtual machines, which allowed mainframes to run multiple applications and processes at the same time. Today the increased computing capabilities of modern hardware has shifted virtualization beyond plain system partitioning. Virtualization today may provide various features like emulation, optimization, translation, isolation, replication, and so on [SN05]. We are especially interested in the isolation aspects of virtualization in order to maintain various protection domains. Virtualization seems to be a viable technology to be applied in desktop systems, enterprise systems, service providers, production systems, virtual private networks, agile and cloud computing, mobile and embedded devices, and other systems. In the last ten years lots of work went into improving the performance, enhancing the flexibility, and increasing the manageability of virtualization technologies [Kro09].

Virtualization can bee seen as some kind of system abstraction of real machines which provide a virtual platform for running tasks. A virtualization layer logic, which is interposed between the hardware layer and some sort of client layer running above it, provides virtualized system resources to the clients (*guests*) [DG09]. The clients gain access to resources via standard interfaces. These interfaces do not communicate with the resources directly but via the virtualization layer, which manages the real resources and possibly multiplexes them among multiple clients. The virtualization layer in general runs at a higher privilege level than the clients, and is able to intercept important instructions and events and treat them in a special way before they are handled or executed by the hardware. Let's assume that a client wants to execute an instruction on a virtual machine. The virtualization layer may intercept that instruction and implements it in a different way on the real resources he has control of. Through this management performed by the virtualization layer, each client is presented with the view of being the only user accessing the resources. The virtualization layer must maintain this illusion and ensure the correctness of the resource multiplexing. So we can say that virtualization is a technique to provide efficient resource utilization via sharing among clients. It also maintains strong *isolation* between clients, that need not know of each other's existence. The abstraction decouples the client from the real resources. This results in greater architectural flexibility and mobility in system design.

3.4.2 Interfaces, Abstraction, and Virtualization

Due to their complexity, computer systems are designed as hierarchies with *well-defined interfaces* to separate levels of abstraction. Examples for well-defined interfaces are the *Instruction Set Architecture (ISA), Application Binary Interface (ABI)*, and *Application Programming Interface (API)*. The ISA is located at the lowest level in the system and directly communicates with the hardware. The other interfaces are located at higher levels and only state a higher degree of abstraction, but in the end all software is executed through the ISA. Figure 3.7 shows where the different interfaces are located in a computer system architecture. Through the abstraction of these interfaces the lower-level implementation details are hidden from hardware and software designers, which as a result simplifies the design process.

The ISA of a computer system examplifies the advantages of a well-defined interface. Software that is compiled for one ISA (e.g. x86), will run on any processor that implements the x86 instruction set. The disadvantage however is that the software is tied to a specific ISA and dependent on a specific operating system interface, if it is distributed as compiled binaries. Virtualization is a technique that can come up with such disadvantages,

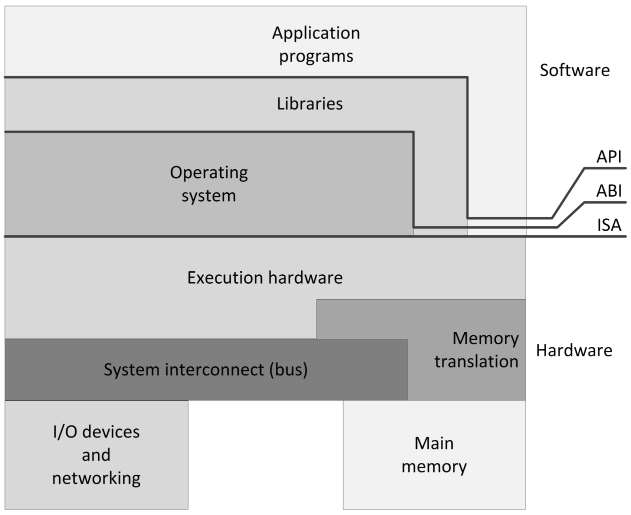


Figure 3.7: Computer System Architecture - Interfaces

since it maps an interface at a given abstraction level onto the interface and resources of an underlying, possibly different, real system [SN05]. The inteface levels like ISA, ABI, API therefore represent an opportunity where virtualization may take place. We will have a look at some virtualization approaches in the following.

3.4.3 Process Virtual Machines

From the view of a process that is executing a user program, a *machine* consists of a logical memory address space that belongs to the process, with user-level instructions and registers that allow the execution of the code that belongs to the process. Processes can only communicate with the machine's I/O through the operating system. Thus the ABI defines the processes' low level view of the machine, the API the high level view. A *process virtual machine (VM)* provides a virtual ABI or API environment through which every process interacts with the operating system and being completely unaware of the activities of other processes [SN05]. Therefore a process VM is a virtual platform that executes an individual process.

The most common process virtual machine is implemented in almost every modern operating system. The operating system virtualizes the memory address space, central processing unit (CPU), CPU registers, and other system resources for each running process.

The operating system is able to support multiple user processes through multiprogramming. It is responsible to guarantee each process a fair time-share of the CPU. Therefore a scheduling algorithm implemented in the operating system context switches through the different processes, maintaining the illusion that each process has sole access to the CPU. Virtualization of memory gives each process the illusion of its own address space, thereby a proccess is not able to access the address space of another process. Virtual memory spaces are achieved through the usage of a *page table mechanism* that translates the virtual memory pages in a processes' virtual address space to actual physical memory pages [DG09]. Usually there are lots of so called *page table walks*, so the address translation is often aided by additional hardware in the processor, like a *Memory Management Unit (MMU)* which is often coupled to a *Translation Lookaside Buffer (TLB)*, and multiple cache levels (*L1- ,L2-cache*).

Process virtual machines may also support the execution of processes that are compiled for a different ISA than the host system. Then the virtualization layer must support the translation from one ISA to another. Two popular techniques for ISA translation are *interpretation* and *dynamic binary translation* [SN05]. A process virtual machine approache that supports different ISAs would be the *Java VM architecture*. Process virtualizaiton enforces *isolation* policies through the runtime component (virtualization layer) which runs the processes under its control. The isolation is guaranteed since the virtualization layer restricts direct access to the underlying real system.

3.4.4 System Virtualization

From the view of an operating system and its supported applications, the whole system runs on an underlying *machine*. A system is a full execution environment which is able to support numerous processes running in parallel. It is responsible for allocating real memory and I/O resources to the processes and manages the interaction between processes and their resources. So the systems' perspective of a machine is defined only through the characteristics of the underlying hardware. Here the ISA provides the interface between the system and the machine [SN05]. A *system VM* is then built by virtualizing the entire system. Therefore the virtualization software, often called *Hypervisor* or *Virtual Machine Monitor (VMM)*, virtualizes all resources of the real machine including the CPU, memory, processes, network devices, graphic cards, and so on. So the hypervisor provides the software access to real resources via a virtualized ISA, giving the software in the virtual machine the illusion of running in a real machine. The hypervisor must run at the highest privilege level in order to be in absolute control of the whole system resources, which is often achieved through processor hardware mechanism like *protection rings*, *privilege rings* or *processor mode mechanisms*. They are used to protect the hypervisor from the guest VMs and to protect VMs from each other.

System virtual machines are full replica of the underlying hardware platform enabling them to run complete operating systems within them. It may also be possible that the hypervisor supports multiple virtual machines. Therefore it is responsible for properly sharing and managing real machine resources among the different virtual machines. All accesses to hardware resources must be mediated by the virtual machine monitor, which as a result provides the necessary isolation between the virtual machines [VN09]. It may also be possible that the hypervisor emulates the hardware ISA, enabling the execution of software with different ISA.

Let us once again have a look at privilege rings. The x86 CPU architecture for example provides 4 protection rings from level 0 to 3 in which the code can execute. In general operating systems are running in ring 0 which has the highest privilege level. All the code running in this ring is often said to run in *system space*, *kernel mode*, *privileged mode* or *supervisor mode*. Other applications that are executed by the operating system run in less privileged rings, normally in ring 3. We heard before that the hypervisor needs to run in the highest privilege mode, therefore occupying ring 0 of the CPU. So the guest operating systems kernels that are running on the system need to execute in less privileged rings. But most operating system kernels are explicitly written to run in ring 0, because they must be able to perform tasks that are only available at this privilege level. Such tasks may be the execution of privileged CPU instructions or the direct manipulation of memory. Two approaches to solve this problem are full system virtualization and paravirtualization that are examined in the following two chapters.

3.4.4.1 Full System Virtualization

In the *full system virtualization* approach the virtual machine monitor (hypervisor) sits directly on top of the hardware and completely simulates the system environment (figure 3.8). All important features of the hardware like the full instruction set, interrupts, memory access, input/output operations are reflected into one of several virtual machines.

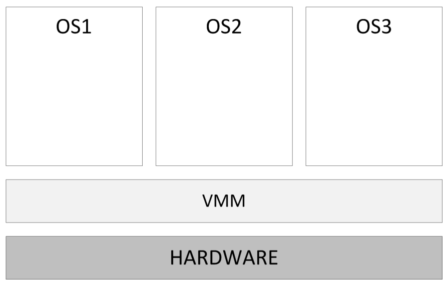


Figure 3.8: Full-system virtualization

Through full system virtualization, any software that is capable of running on the hardware can be executed in a virtual machine, which also includes the support to host *unmodified* operating systems. The term unmodified means that the operating system kernels, in comparison to *paravirtualization* (see chapter 3.4.4.2), need not to be modified in order to run on the hypervisor. Therefore privileged operations still run in ring 0 of the CPU. This privileged instructions may potentially alter the state of any other virtual machine, the hardware, or the control program. So the main purpose of the hypervisor is to intercept and simulate privileged operations made by unmodified operating systems through CPU emulation. Compared to paravirtualization, one of the major drawbacks of full system virtualization is that this emulation process requires both time and system resources which may result in weak performance.

3.4.4.2 Paravirtualization

In the *paravirtualization* approach, the kernels of the guest operating systems need to be *modified* in order to run on the hypervisor (see figure 3.9). This modification presents a software interface to the virtual machines which may differ from that of the underlying hardware. Through the modification, the privileged operations and instructions (e.g. calls to hardware and device drivers) that will only run in ring 0 of the CPU are replaced by calls to the hypervisor (also known as hypercalls). The hypervisor then carries out the task on behalf of a privileged guest operating system that may host device driver and hardware functionality. Through this approach it is possible that software running on a virtual system can bypass the virtual interface and actually carry out it's operations on the system's hardware directly resulting in better peformance. This is because in full system virtualization all the hardware is utilized by the virtual interface, but some of the privileged functions are very difficult for the virtual system to accomplish. Therefore

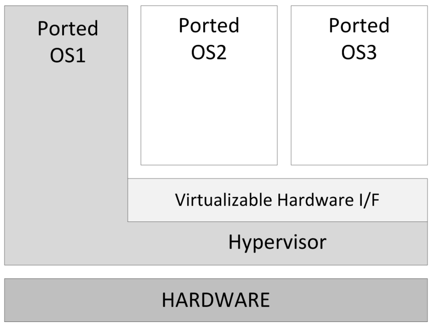


Figure 3.9: Paravirtualization

a virtual program that needs to carry out one of these tasks takes less resources when it skips the virtual layer and goes directly to the hardware system. A major drawback of paravirtualization on the other hand is the lack of flexibility. Since the kernels of the guest operating systems need to be ported, paravirtualization is often limited to support open source operating systems like Linux, porting proprietary kernels like Windows is mostly not possible. A very popular paravirtualized environment is *Xen* [BDF$^+$03].

3.4.5 Shared Kernel Virtualization

Shared kernel virtualization is often also referred to as operating system virtualization or system level virtualization. It is based on the architectural design of Linux and UNIX operating systems. This design is based on two main components. The first component is the kernel. As we know, the kernel manages the interactions between the physical hardware and the operating system. The second key component is the root file system. The root file contains all the files, libraries and utilities that are necessary that the operating system is running. Now in the shared kernel virtualization approach, the virtual machines (also called system partitions) all have their own root file system but share the same kernel of the host operating system (see figure 3.10). This is achieved through an enhancement of the *chroot* feature of the operating system, where a complete system partition is created by chrooting a standalone machine with its own root file system. A drawback of this approach may be, that guest operating systems must be compatible with the shared kernel of the host system. Example systems are *Solaris Zones, Virtuozzo, OpenVZ.*

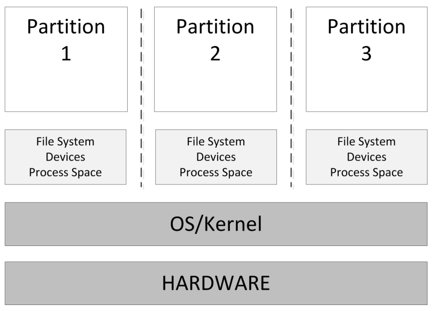


Figure 3.10: Shared kernel virtualization

3.4.6 Kernel Level Virtualization

In this virtualization technique a host operating system runs on a modified kernel that has extensions to manage multiple virtual machines. These virtual machines all contain a guest operating system. In comparison to shared kernel virtualization each guest runs its own kernel. Modifications to Linux kernels for example may be implemented as *kernel loadable modules* (see chapter 5.4.6). Though shared kernel and kernel level virtualization are different, the restriction that guest operating systems have to be compiled for the same hardware as the kernel they are running upon remains. Examples of kernel level virtualization technologies are *Kernel-based Virtual Machine (KVM)* and *User Mode Linux (UML)*.

3.4.7 Guest Operating System Virtualization

Guest virtualization is probably one of the easiest virtualization concepts. Here the host computer system is running a standard unmodified operating system like Windows, Linux or MacOS. Then a third-party virtualization software, running on the host system like a normal application, creates one or more virtual machines that run guest operating systems on the host computer. It is responsible for starting, stopping and managing the virtual machines and to regulate the accesses to physical hardware. In this approach the guest systems are fully virtualized and are not aware of other guest systems. They believe that they are the only running system with their own hardware. This effect is enforced by the virtualization application through a binary rewriting process. This process scans the application stream of the executing guest system and replaces privileged instructions with safe emulations. This high level of abstraction of the emulation process between the guest operating system and the host hardware sometimes ends op in weak virtual machine

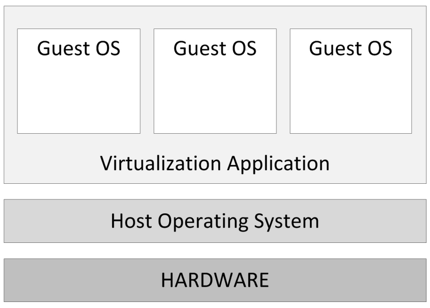


Figure 3.11: Guest Operating System Virtualization

performance. Especially disk I/O often is carried out slowly. However, the speed of nondisk operations is near native. Therefore it is recommended to use network services to communicate between different virtual machines like Windows Terminal Services (RDP) or SSH in Linux and UNIX systems. On the other hand a big advantage of this technique is, that no changes have to be made either to the host or guest operating systems and that no special CPU hardware virtualization support is required. Examples of guest OS virtualization technologies would be *VMware Server* and *VirtualBox*.

3.4.8 Hardware Virtual Machines

A *hardware virtual machine* is built by using primitives that are provided directly through the hardware like I/O or the processor. Compared to the software solutions like process or system VM this approach has the advantage of huge performance improvements. Hardware virtualization in gerneral also provides better isolation between the virtual machines which results in higher security. An example is the *Intel Virtualization Technology* [NSL$^+$06].

3.5 Operating System Kernels

The operating system kernel has always been the most trusted component in a computer system. So it is apperant to entrust the kernel with enforcing isolation between application and applications and the kernel itself, which actually is the most traditional isolation approach. In the last years much research was spent on reducing the size of the kernel since a smaller kernel also reduces the size of the TCB. As a result and opposed to monolithic kernels so called *Microkernel* and *Exokernel* operating systems have emerged where the code of the kernel consists of some thousand lines of code only, which makes them easier to be proven correct.

Microkernels especially gain interest in the embedded systems domain. Traditional embedded systems software was build on top of a real-time execution environment without memory protection. Then the complete system, which may contain out of millions of lines of code belongs to the TCB. In such a system every bug in any part of the system can cause a security violation. It's clear that it is impossible to make such a large TCB trustworthy. However, in modern embedded systems a memory protection mechanism is almost always integrated. Hence it is necessary to have a operating system that is able to deal with those memory protection mechanisms. This started the trend to run commodity operating systems like Linux and Windows on the embedded system. Those are often stripped down versions of their originals, but still have sizes of around 200.000 lines of code (LOC), potentially containing hundreds, if not thousands of bugs. To secure the system we need a secure and *trustworthy* TCB, which basically means it has to be free of bugs. To show the correctness of this code we could use exhaustive testing ore formal methods like mathematical proofs, but those scale poorly and are typically limited to hundreds, at most thousands of LOC. Therefore another approach is to modularize the system in order to deal with the complexity and to separate the problem in more tractable segments. Modularizing the kernel code does not solve our problem, because kernel code is running in the privileged mode of the processor, so there is no protection against kernel code that is violating module boundaries. The kernel code is *atomic*. So we need to modularize non-kernel code, in such a way that the modules can be encapsulated into their own address spaces. This actually means that hardware mechanisms enforce the module boundaries under the mediaton of the kernel. Then the trustworthiness of such a module can be established independently from other componets, if the kernel is trustworthy.

So how to actually make the kernel trustworthy? As we already heard, the answer to this question lies in reducing the TCB which actually is done by reducing the size of the kernel, maintaining a microkernel. The microkernel must be small enough so that it can be verified. To achieve this goal, the microkernel only contains code that must run privileged. Any other unprivileged code should stay outside of the kernel, i.e. run in user mode. The microkernel then is the near-minimum of code that is able to provide the mechanisms needed to implement an operating system. Such mechanisms would

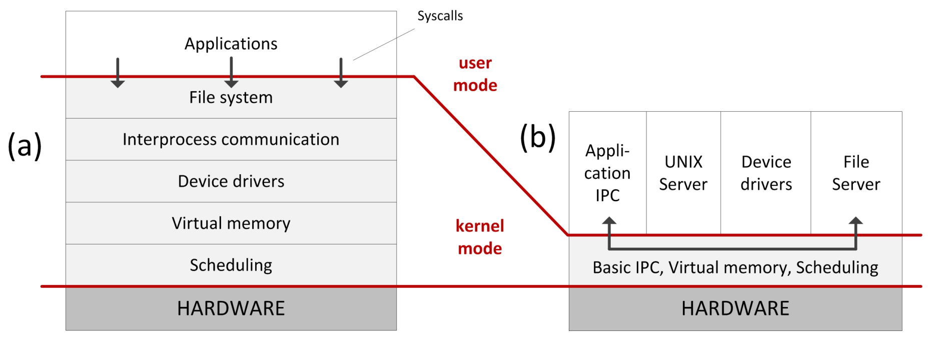


Figure 3.12: (a) Monolithic kernel and (b) microkernel

be thread management, inter-process communication (IPC), and low-level address space management. Other traditional functions of operating systems such as protocol stacks, file systems, and device drivers are then removed from the microkernel and run in user mode (see figure 3.12).

The large size of the TCB of traditional operating systems is one reason that they are impractical for the requirements of embedded systems. At least a second reason is that they have a model of access control that only protects different users of the system from each other. But there is no possibility to restrict the access of particular users to their own data. Most embedded systems are single-user systems, so the protection issue is different to the desktop environment. There should be the possibility that different programs of the same user can have different access rights to system resources. Those access rights shouldn't be based on the identity of the user but rather by the function of the programs. This is an instance of the security *principle of least privilege* (see chapter 5.4.2). Because traditional systems run programs with the full set of the user access rights the principle gets violated. Through this violation viruses and worms could cause much damage in the system. Imagine that a game installed on your system has an embedded virus. Running the game under the full user privileges may allow the virus to destroy files or steal confidential data.

In a system with a microkernel approach, we heard that the software is encapsulated into modules where the module boundaries are maintained with hardware-enforced interfaces. So every communication between modules must employ the kernel-provided IPC mechanism, which means that the kernel is in full control over all communication between components. This offers the opportunity to interpose security monitors between components. These security monitors may be used to enforce system-wide security policies.

These policies could be used to allow an imported program (such as a game) only access to files that the user assigned to it. This approach often is referred to as *fine-grained access control*. Example microkernels are the *Mach Microkernel* [ABB+86] and *L4*.

3.6 Hardware-based Isolation

Hardware-based isolation is the strongest form of isolation, which is quite difficult to be circumvented by the software at runtime. Such mechanisms may be provided by the processor itself, or by special hardware that works together with the processor, like a memory management unit we already heard about. There may also be so called I/O Memory Management Units, that translate the Direct Memory Access (DMA) address of a device to physical addresses. The I/O MMU allows a device only access to memory regions that it has been explicitly granted to. So malicious devices are prevented from performing arbitrary DMAs, which increases the system availability and reliability. An examples is the *Calgary I/O MMU* [BYMX+06]. Another hardware-based isolation approach is the *ARM TrustZone* [Yor03] technology (see chapter 2.2).

CHAPTER 4

Case Study: Key Distribution Tool

4.1 Introduction

Almost every security service of a system is based on a cryptographic mechanism like encryption or authentication. These mechanisms need cryptographic keys that must be distributed to communicating parties prior to secure communications. So if we want to integrate cryptographic functions into our systems we need a secure key management facility that is able to provide secure procedures for handling crpytographic keying material. According to the *Open Systems Interconnection (OSI) Security Architecture Standard ISO 7498-2* [ISO88], key management is "the generation, storage, distribution, deletion, archiving and application of keys in accordance with a security policy". Secure key management is one of the most critical tasks when integrating cryptographic functions into a system. Because of this high criticality it is also very important to be aware of the possible threats to protect against and to know about the physical and architectural structure of the system [FL93].

One of the central problems in key management is the secure storage and distribution of keys. It is necessary to guarantee the origin, integrity, and confidentiality of the keying material. Therefore we will define and implement a *Key Distribution Tool (KDT)* which serves as the basis for our evalution work in chapter 5.

4.2 Functionality

The implemented Key Distribution Tool consists of the *Key Distribution Center (KDC)*, the *Secure Key Distribution Protocol (SKDP)* and *User Programs*. User programs may request new or existing keys from the KDC using the SKDP.

If a user program requests a new key, the KDC uses a Pseudo Random Number Generator (PRNG) to generate a cryptographic key. Then the key is stored in the key database and afterwards sent back via the SKDP to the requesting user program. The key database is implemented as a hash table where the hash index is calculated over a unique *key_id* that is sent within a key request from user programs. If the requested key already exists in the database it will be overwritten with the newly generated.

If a user program requests an existing key, the KDC looks it up in the database and if found, the key is sent back to the user program. If there is no such key in the database, the KDC again uses the PRNG to generate one, then writes it to the database and sends it back to the user program.

On every key request, the SKDP automatically initiates a key exchange based on the *Diffie Hellman* protocol, where the KDC and user programs are supplied with a shared secret key. The calculation of the shared secret is done by using *Elliptic Key Cryptography (ECC)*, where ECC keys in general are harder to attack than e.g. symmetric keys. The shared secret key then is used by the KDC to encrypt the key messages, thereby securing the communication. The user programs then of course use this shared secret to decrypt the received message. Encryption and decryption is done be using the symmetric AES cipher. All of the cryptographic stuff in the KDT is done by using the libraries *LibTom-Crypt (libtomcrypt)* and *TomsFastMath (libtfm)* found under [SD10].

4.3 Block Diagram

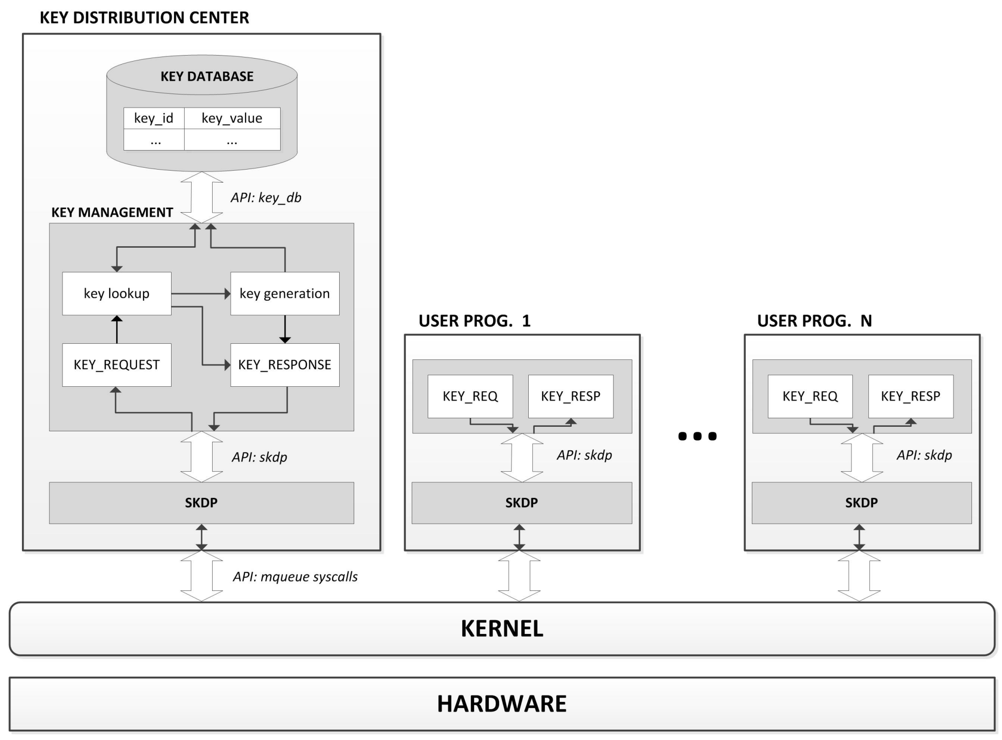


Figure 4.1: Schematic Overview of the Key Distribution Tool

4.4 Secure Key Distribution Protocol

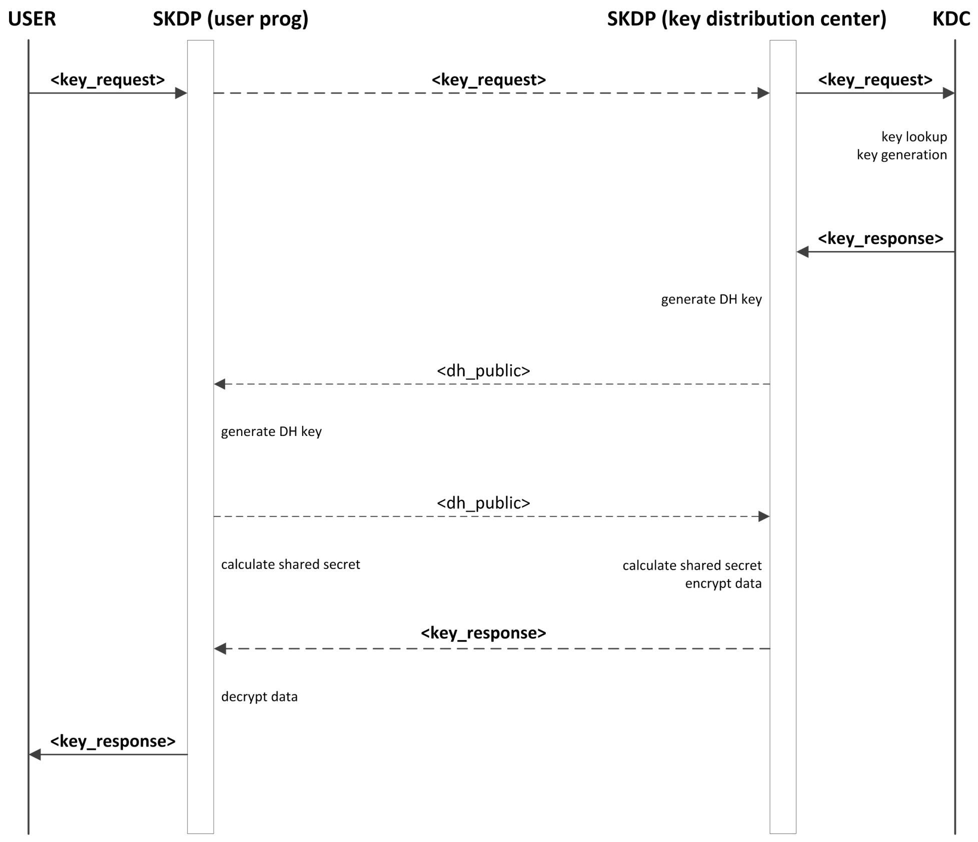


Figure 4.2: Diagram of the Secure Key Distribution Protocol

CHAPTER 5

Evaluation

5.1 Evaluation Goals

In the previous chapters we had a look at basic security-related topics to gain a first insight to this huge area. We concluded that there is a wide design space from the pure software and hardware solutions up to hybrid approaches for implementing security features. We figured out that isolation techniques play an important role for the secure execution of software, making the system more error-prone and less vulnerable to attacks by malicious users. Next we implemented the Key Distribution Tool (KDT) as a classical example for security concerned software. In this first step the KDT was implemented to run on a monolithic Linux kernel using its POSIX interfaces for memory access and interprocess communication. This decision was based on the fact that due to its low cost and ease of customization there are lots of derivatives of the Linux operating system (e.g. *Embedded Linux, μClinux*) that are actually running on many of today's embedded devices (see chapter 3.5). Since the Key Distribution Center (KDC) handles cryptographic keys and stores them in main memory, it has to be isolated from other processes in order to prevent the exploration of the sensitive key information. So one of the first steps of the evaluation process will be to examine how the Linux kernel enforces this isolation. Then, in a case study, we will try to find mechanisms that still make it possible to explore the keys. In addition we will have a look at Linux features that probably are inappropriate for the embedded systems domain, as for example the Linux Discretionary Access Control model.

The next major step in the evaluation process will form the core statement of this work, which is a comparison between the Linux operating system and a *Partitioning Operating System* approach. Partitioning Operating Systems are especially designed to meet safety and security requirements in a time-critical embedded system environment. In our case we will evaluate *SYSGO's PikeOS* [SYS12]. Therefore we will give an insight to the architecture of this modern real-time operating system. Afterwards we examine the situation for secure key storage in PikeOS by comparing the cases for memory acquisition found in Linux. Through this we will be able to work out the architectural benefits of a partitioning operating system, which in the end allows us to formulate the conclusion about this work.

5.2 Use Case

First of all we will define a use case upon the KDT that will be used throughout the whole evaluation process. One possible utilization of the KDT could be that applications request for keys in order to encrypt data they want to send to other applications locally or even over some network interface. Another example for an implementation using the KDT could be a "*Secure Group Communication Service*" where applications share a cryptographic key as a common secret. Then all applications in posession of the shared secret are allowed to participate in the group communication. In both cases there will also be applications that are not allowed to know about those keys, which forms the major security requirement for our use case. Therefore we need to distinguish between both sorts of applications, calling them *trusted* if they are and *untrusted* if they are not allowed to request for cryptographic keys.

Refining our use case to a concrete scenario, we will allow two applications to communicate with the KDC forming a *trusted partition*. All the other applications should be rejected, hence forming an *untrusted partition*. The KDC along with the core messaging services are part of the *Trusted Computing Base* of the system (see figure 5.1). Then we will implement the use case on both of our evaluation systems: *Linux* and *PikeOS*. To examine how strong both systems enforce separation between trusted and untrusted domains, we will define a debug key and try to explore this key from the KDC's key database. Since allocated dynamically, the key database will be located in heap memory. The value of the debug key is *"00112233445566778899AABBCCDDEEFF"* and will be inserted into the database when starting the KDC.

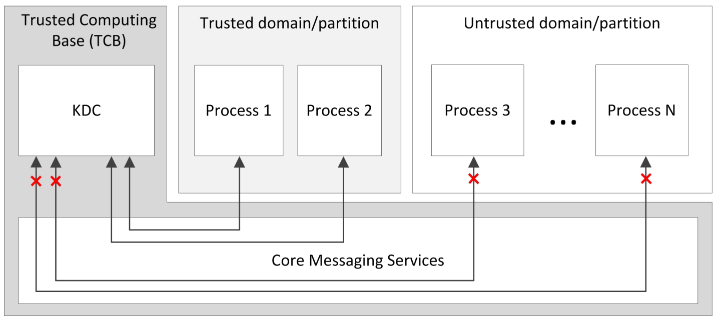


Figure 5.1: Key Distribution Tool Use Case

5.3 Linux

5.3.1 Interfaces to Linux

As said before, our first evaluation environment is Linux and the Linux kernel. To get familiar with Linux we will examine some architectural features in the following chapters. Linux is based on a monolithic kernel architecture which is compliant to the *Portable Operating System Interface (POSIX)* standard. POSIX is a family of standards specified by the IEEE for maintaining compatibility between operating systems described in the *IEEE 1003.1-2008 Standard* [Ope12].

The Linux kernel runs on top of the bare hardware and provides a *system call interface* to all the running user programs. The system calls may be used by user programs for example to create, destroy and manage processes, files, or other resources. System calls are done by placing arguments in registers or on the stack, and then issuing a trap instruction into the operating system to switch from *user* to *kernel mode*. Those two modes of operation are provided by most of the ISAs in use and are often also referred to as *privileged mode* and *unprivileged mode* (see chapter 5.3.2). Trap instructions cannot be written in C, so a library containing one procedure per system call is provided. Therefore the POSIX standard specifies a library interface and not the system call interface itself. So making a system call is like making a procedure call, with the difference that system calls enter the kernel and procedure calls do not. The kernel code then starts after the trap instruction by examining the call number. This number then will be dispatched to the correct system call handler. After processing the system call handler code the execution control is given back to the user code.

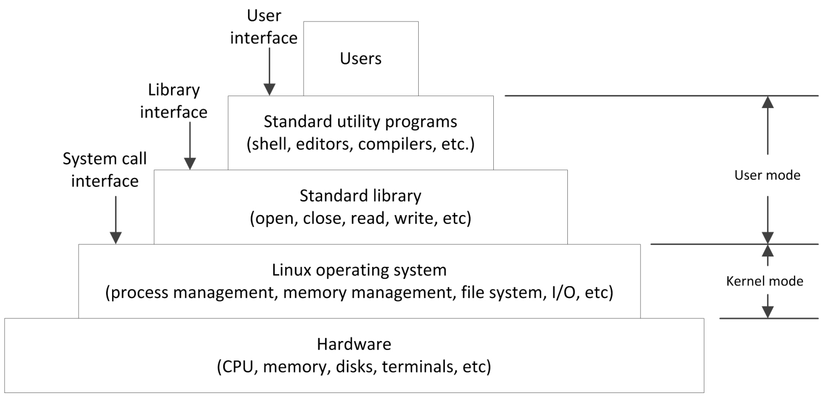

User interface
Users
Library interface
Standard utility programs (shell, editors, compilers, etc.)
System call interface
Standard library (open, close, read, write, etc)
Linux operating system (process management, memory management, file system, I/O, etc)
Hardware (CPU, memory, disks, terminals, etc)
User mode
Kernel mode

Figure 5.2: Linux layers [Tan07].

Besides the operating system and system call library the POSIX standard also supplies a number of user level programs specified by the *POSIX 1003.2 Standard*. Some of those programs include the command processor (shell), compilers, editors, text processing, and file manipulation programs. Thus POSIX consists of three different interfaces to Linux: the *system call interface, library interface*, and the *user interface* that is formed by the set of standard utility programs (see figure 5.2).

5.3.2 The Kernel

The Linux kernel directly sits on top of the hardware and is responsible for interacting with the memory managment unit, I/O devices and also controls the CPU access to them. On the lowest level of the kernel sit the interrupt handlers that are primarily used to interact with the devices and low-level dispatching mechanisms. The kernel runs in the *privileged mode* where it has complete access to all the hardware and instructions of the CPU. All the other software runs in the *unprivileged mode* where only a subset of the machine instructions is available. Privileged and unprivileged modes are enforced by the protection rings of the CPU (see chapter 3.4.4). The Linux kernel consists of three main components:

- **Input/Output:** In this component all code for interacting with devices and performing network and storage I/O operations are located. We can see on top, that the I/O operations are all integrated under a virtual file system layer. This means that a read operation, that may be used for reading files from memory or disk, or to

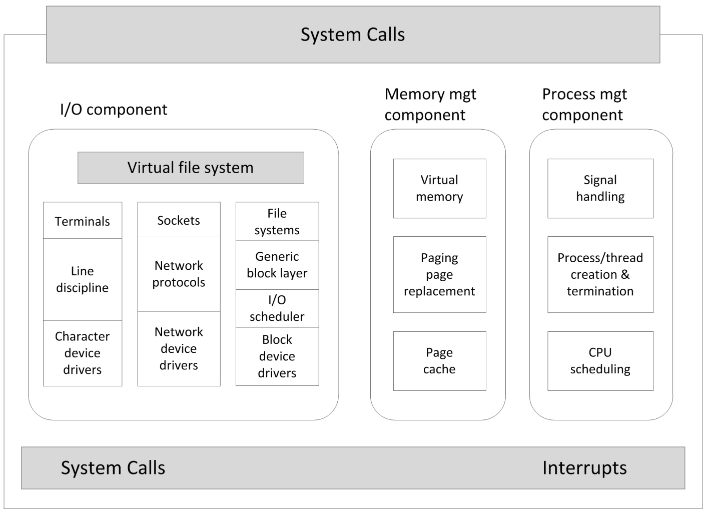


Figure 5.3: Structure of the Linux Kernel [Tan07].

retrieve a character from a terminal input, remains the same. On the bottom level the I/O operations always pass through a device driver.

- **Memory Management:** In the memory management component all the mappings from virtual to physical memory are maintained. Other tasks include page caching and page replacement, and on-demand paging.
- **Process Management:** The main task of the process management component is to create and terminate processes. The process scheduler of course is also implemented in this component and responsible for scheduling processes and threads based on a global scheduling policy. Linux treats both processes and threads simply as executable entities.

The three main components of the Linux kernel are statically compiled into the kernel. But Linux supports an opportunity to dynamically load code into the kernel, called *Loadable Kernel Modules (LKMs)*. LKMs may be used to replace the default file system, networking, device drivers, or other kernel codes. We will have a look at LKMs later on in the evaluation process, since they may introduce a serious threat to the operating system's security (see chapter 5.4.6). At the highest level sits the system call interface into the kernel. At this interface the incoming system calls cause a trap that switches the execution from user mode into kernel mode. The control then will be passed to one of the above kernel components [Tan07].

5.3.3 Processes

5.3.3.1 Process Isolation and Virtual Memory

In chapter 3 we heard about isolation techniques for secure software execution. Most operating systems including Linux enforce isolation with the CPU's memory management hardware (MMU) at process granularity. One of the first steps for a secure and stable system is the protection of the kernel from other running programs, which is achieved by the two modes of execution of the ISA. But this alone is not enough, since we also need a mechanism to protect running programs from each other as well. This is achieved by the Linux's *virtual memory subsystem*. It is responsible for managing the translation from virtual to physical addresses and vice versa, thereby enforcing separation between the processes' *address spaces*. The kernel actually sets up the virtual memory system at the software level. There the kernel ensures that it is able to access the address space of any process. The address space of a process is it's range of virtual memory addresses. The kernel also makes sure that no process can directly reference the kernel memory. With that, virtual memory gets devided into two areas, often referred to as *kernel-land* and *user-land* memory (see chapter 5.3.3.2). More details on virtual memory and memory management may be found in [Tan07].

5.3.3.2 Processes in Memory

Since we want to explore keys from the memory of the runnig KDC process, we will now examine in short how processes in Linux actually reside in memory. When running Linux on a 32-bit machine, every process has a virtual address space of 3 GB. The only way processes can communicate with the underlying operating system or with other processes is via system calls. When issuing a system call the process traps to the kernel and runs in kernel context from that point on. In kernel context each process has access to an additional 1 GB of kernel memory, that is not visible to the process when in user mode. Each process is described by a process descriptor struct called *task*, which actually is resident in the kernel memory all the time. This struct presents information for the kernel to manage all the processes. The information in the task struct can be catogorized into scheduling parameters, memory image, signals, machine registers, system call state, file descriptor table, accounting, kernel stack, and miscellaneous data.

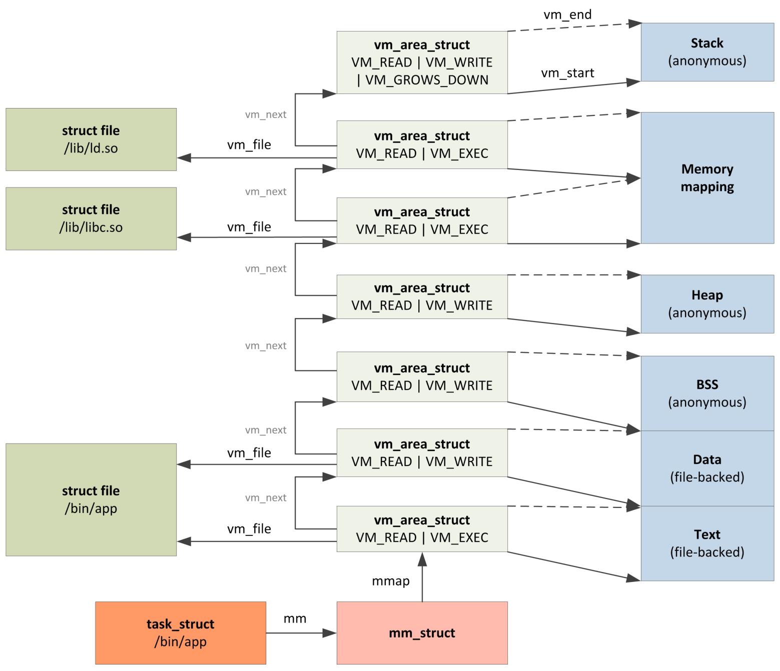


Figure 5.4: Linux Virtual Memory Structure.

In the memory image field resides the virtual memory information of the process, containing pointers to the text, data and stack segments or virtual pages. All this information is stored in the top-level memory descriptor called *mm_struct*, also known as *page table*, which is created for each process. On a context switch, the page table belonging to the corresponding process will be loaded. Because every process has its own page table it also has a different set of pages than the others, and therefore sees a large, contiguous, virtual address space all for itself. Information to virtual pages are stored in the *vm_area_struct* structure (see figure 5.4). If we want to access the memory of the KDC we somehow will be confronted with system calls and kernel mode and the *mm_struct* structure (see chapter 5.4).

5.3.3.3 Communication

Processes in general can communicate with each other over either *message passing* mechanisms or by sending *signals*. Linux offers *pipes, message queues, shared memory*, and *semaphores* for message passing. We already dealt with message queues in the implementation of the KDT (see chapter 4).

Signals can be sent from one process to the other. Each process can tell the system what he wants to do if a signal is received by choosing if he wants to ignore it, catch it, or let it kill the process. Killing the process is the default operation for most signals when nothing else is defined. If a process catches a signal he needs a signal handling procedure. Signal handlers are handled like hardware interrupts. On receiving a signal, the control is instantly transferred to the signal handler. When it finished it is given back to where it came from. In Linux processes are only allowed to send signals to processes in their own group. A *superuser* may send signals to *all* processes on the system (see chapter 5.3.4 and 5.4.2.1). The POSIX standard defines system calls for message passing and sending signals and a set of appropriate signals. It also allows to change the signal mask of a process, whereas the signal mask defines a set of signals that are currently blocked on the receiving thread.

```
int sigprocmask(int how,
                const sigset_t *set,
                sigset_t *oldset);
```

- `int how`:
  Defines behavior when calling *sigprocmask()*:
  - SIG_BLOCK:
    Set of blocked signals is the union of the current set and the *set* argument
  - SIG_UNBLOCK:
    Signals will be removed from the current set of blocked signals.
- `const sigset_t *set`:
  If *NULL* the signal mask remains unchanged, otherwise *set* becomes the current set.
- `sigset_t *oldset`:
  If not *NULL* the previous signal mask will be stored in *oldset*.

5.3.4 Security in Linux

5.3.4.1 Access Control

UNIX-based operating systems including Linux are so called *multiuser systems*. The basic line of defense of a Linux operating system is its per user based *Discretionary Access Control (DAC)* mechanism (see chapter 3.3.3), which determines if a entity (user or program) can access a given resource. Every Linux system consists of a set of registered users who are assigned a unique integer called *User ID (UID)*. Users can also be organized into groups which are identified by a *Group ID (GID)*. Upon creation, all files, processes, and other resources inherit the UID, GID, and access permissions of the creator, also referred to as the *owner*. The access permissions determine who is able to access the resource. They are split up into three sets of users, the group, and others. We will see that the per user based access control mechanism of UNIX-like systems is impracticable for enforcing separation in the embedded system domain (see chapters 5.4.1, 5.4.2.1, and 5.4.4).

In Linux the special user with UID 0 is called *superuser* or *root*. It is responsible for all the administrative tasks like managing others users, configuring the system, setting usage limits, installing device drivers, and so on. Therefore a superuser has full privileges, allowing him to access all resources and using protected system calls. It is also allowed to modify the kernel itself, since like any other software the kernel needs to be updated too. So if a process reaches superuser status it has full control over the machine. Actually most of the system attacks aim at reaching this status which is often referred to as *privilege escalation* (see chapter 5.4.2.1).

5.3.4.2 Security Features

DAC of the Linux kernel alone is insufficient for providing security (see chapter 5.4.2.1). Therefore there are lots of projects out there aiming at enhancing the security in Linux. It has been shown, that it is most effective to place countermeasures and security features in the kernel, often implemented as *loadable kernel modules* (see chapter 5.4.6).

One of those projects is *grsecurity* [GRS12] which is a set of security patches for the Linux kernel. One important part of *grsecurity* is that it allows to define a *least privilege policy* for the system, where every process and user only has the lowest privleges they need to function (see chapter 5.4.2.1). Another major component in *grsecurity* is the *PaX* project, which provides *Address Space Layout Randomization (ASLR)*. Through ASLR every process gets a random address space, making the system somehow indeterministic and preventing it from attacks by buffer-overflows.

A further example is *Security Enhanced Linux (SELinux)* which is using the *Linux Security Modules (LSM)* interface [SEL12]. The LSM interface provides everything needed to implement a *Mandatory Access Control* module while doing only a few changes to the Linux kernel. Mandatory Access Control has always been closely associated with multi-level secure and trusted systems (see chapter 3.3.3 and 2.1.4).

Some other improvements introduced by other security features are *Media Access Control (MAC), Access Control Lists (ACL)*, and *Role-Based Access Control (RBAC)*. Moreover, enhanced Linux security features restrict the usage of *ptrace* (see chapter 5.4.5), *chroot*, and access to diagnostic interfaces like the */proc* pseudo filesystem.

5.4 Memory Acquisition

5.4.1 Assumptions

Through the steady security enhancement of the Linux kernel it is ever getting harder to attack Linux operating systems. Therefore finding and writing new exploits and attacks is a time-consuming act where the attacker needs creativity and serious programming skills. Exploiting the kernel or driving user or kernel space attacks would be out of the scope of this work. Another issue is that the security features we heard about before, are just addons to the kernel, making the privileged code even larger and more complicated and have not – as opposed to *PikeOS* – been part of the operating system design. So the goal of our evaluation will be more to compare the design and architectural features between *Linux* and *PikeOS*, without considering additional security features of the Linux kernel. Furthermore, since embedded systems are single user systems, we will assume that besides the superuser there is only one user present making user groups unneccessary. Actually this is one of our main evaluation points. In DAC access to system resoures is always done on a user basis, and not per process which is too course-grained for our use case and the embedded system domain. The multi-user approach just doesn't fit.

The problem will be described again in chapter 5.4.2.1. Also when a user creates new resources like message queues they inherit the access rights of the user. When a user spawns new processes they may also access the message queue, again compromising our use case. To separate those processes we would need to impelement isolation techniques like sandboxing or virtualization, we heard about in chapter 3. But those techniques often lack performance, suffer from race conditions, or have high configuration overhead. Additionally the problem that the large and vulnerable Linux kernel enforces isolation and applies the security policy remains.

5.4.2 CASE 1: Privilege Escalation and Exploitation

Attacking applications or the operating system by software means is often referred to as *exploitation*. Software has bugs which make software misbehave or perform a task incorrectly. Exploitation then means to turn this misbehaviour into an advantage for the attacker. Not all bugs are exploitable, however those that are, are called *vulnerabilities* (see chapter 2.1.2). The monolithic Linux kernel with some million lines of code (LOC) may have ten-thousands of bugs (see chapter 3.5), giving a malicious user a broad range of attack points. However, most of the exploits aim at raising the attacker's privileges to superuser status and therefore gaining access to the whole system (see chapter 5.4.2.1).

5.4.2.1 Privilege Escalation

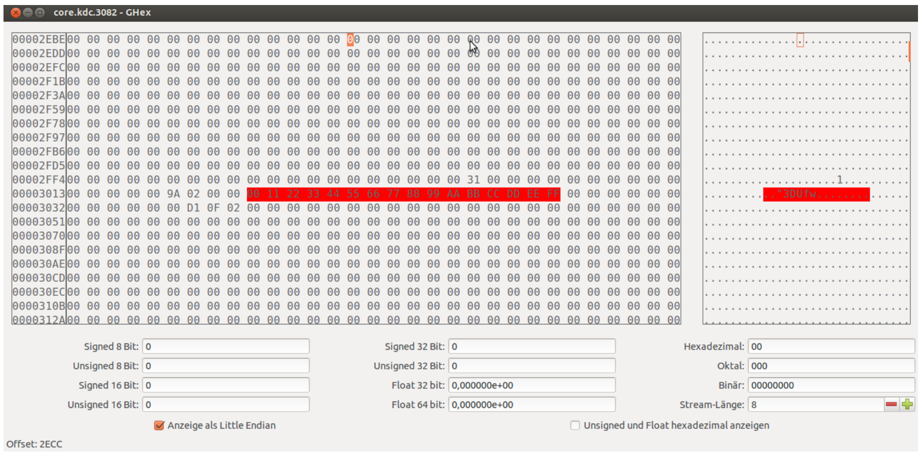
Most operating systems including Linux implement access control via a user model with a privileged superuser (see chapter 5.3.4.1). Although the strength of this user model lies in its simplicity, it has a major drawback: it does not properly capture the usage model of the applications running on a system. Imagine that if a user wants to perform a privileged operation X, he must be designated as the superuser, hence elevating its privileges. The user now may execute other privileged operations besides X, which shouldn't be the case. So this model alone is insufficient for providing security, moreover it is impractical for actually providing *separation* between applications according to our use case. From a security standpoint, this user model can be improved by applying the *principle of least privilege* to the whole system (see chapter 3.5 and 5.3.4.2). This means that the privileges get seperated and that a user only gets the privileges the user needs to perform a specific task. Becoming the superuser does not mean to have full control over the system anymore, since by now the privileges assigned to specific user-land programs decide what they can or cannot do. Hence privilege seperation reduces the amount of code running with full privileges. Techniques like *Media Access Control (MAC), Role-Based Access Control (RBAC)*, and *Access Control Lists (ACL)* enforce the principle of least privilege to the whole system. As we heard in Linux those techniques are enforced by various security features (see chapter 5.3.4.2).

5.4.2.2 Exploits

Besides the various ways a malicious user can reach the status of a superuser, exploiting user-land or kernel-land code is the supreme discipline of any attacker. We heard that this can be a challenging task, requiring serious skills, cleverness, and a lot of dedication. In the user-land there are different attacks like *shellcode injection* and *suid()/guid() attacks*. But through the security features (see chapter 5.3.4.2) it is getting harder to carry out attacks in user-land. Therefore today it is more popular to attack the kernel by exploiting its bugs by e.g. *buffer overflows*. However, there exist lots of user- and kernel-land attacks, stating that Linux and its kernel is actually ***not secure***. We won't introduce exploits since this would be out of the scope of this work, but we will compare between the Linux kernel and the PikeOS kernel on an architectural level concerning attack protection. For now we can say that virtualization systems are becoming increasingly popular, they introduce possibilities to protect the kernel. There already exist virtualization solutions for the Linux kernel too like *XEN* and *KVM* (see chapter 3.4.4).

5.4.3 CASE 2: Sending Signals

In chapter 5.3.3.3 we heard about sending and receiving signals and signal handlers for communication between processes. The POSIX standard defines a set of signals like *SIGINT, SIGKILL, SIGSEGV*, and so on, that must be implemented in an operating system. For completeness you may look up the list on your own. Every operating system defines a default signal handler for each of these signals whereby a user may implement its custom signal handler. On Linux systems signals can be sent between processes by using the *kill()* system call. For some signals like *SIGSEGV* the default signal handling defined by the operating system is to terminate the process that is receiving the signal and then creating a core file (memory image) of the process. The *SIGSEGV* signal is sent to a process when it makes an invalid memory reference or segmentation fault. A process is only permitted to send signals to it's own process group. A superuser can send signals to any process.


core.kdc.3082 - GHex

```
00002EBE 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
00002EDD 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
00002EFC 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
00002F1B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
00002F3A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
00002F59 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
00002F78 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
00002F97 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
00002FB6 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
00002FD5 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
00002FF4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 31 00 00 00 00 00 00 00 00 00 00  ....................1..........
00003013 00 00 00 00 00 9A 02 00 00 00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF 00 00 00 00 00 00  ..........."3DUfw..............
00003032 00 00 00 00 00 00 D1 0F 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
00003051 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
00003070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0000308F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
000030AE 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
000030CD 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
000030EC 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0000310B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
0000312A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............................
```

Signed 8 Bit:	0	Signed 32 Bit:	0	Hexadezimal:	00
Unsigned 8 Bit:	0	Unsigned 32 Bit:	0	Oktal:	000
Signed 16 Bit:	0	Float 32 bit:	0,000000e+00	Binär:	00000000
Unsigned 16 Bit:	0	Float 64 bit:	0,000000e+00	Stream-Länge:	8

[✓] Anzeige als Little Endian   [ ] Unsigned und Float hexadezimal anzeigen

Offset: 2ECC

Figure 5.5: Examining the core file by a hexeditor.

We will not be concerned about access control rights since we saw in chapter 5.4.2.1 that they might be compromised. Therefore in the following steps we will show how we can use signals to produce core files from which we can read the secret key stored in the KDC.

1. On many systems the default core file size is set to 0. So sending *SIGSEGV* wouldn't produce a core file.

   ```
   user@desktop:> ulimit -c
   0
   ```

2. Therefore we will set the core file size to unlimited.

   ```
   user@desktop:> ulimit -c unlimited
   ```

3. Then start the KDC.

   ```
   user@desktop:> ./kdc
   PID: 5797
   ==========
   KEY_DB:
       key_id: 1; value: 00112233445566778899AABBCCDDEEFF
   ==========
   ```

4. Send SIGSEGV to the KDC causing it to terminate and produce a core file.

   ```
   user@desktop:> kill -s SIGSEGV 5797
   ```

After those steps, the core file is created in the directory of the KDC. Using a hexeditor we can now read the file and actually find the key (see figure 5.5). In chapter 5.3.3.3 we saw that POSIX offers the opportunity to block the reception of signals on certain processes. Hence one way to prevent the kernel from producing a core dump would be to block the SIGSEGV in the KDC application. But this would be no good idea, since we cannot distinguish if the signal origins from another user process or from the kernel, stating that there really was a segmentation fault. In this case moving on with normal operation could cause serious problems.

5.4.4 CASE 3: Authorization and Interprocess Communication

This is probably the simplest and shortes case to get the key information from the KDC. As already described in chapter 4, processes can request keys via the the SKDP when calling the function *client_secure_send_key_request()*. The KDC was designed to be simple and small, therefore no authorization mechanism was implemented. This means there is no instance in the KDC that identifies processes by their IDs that determines if a process is allowed to request keys. Through our single user assumption (see chapter 5.4.1) all the processes are allowed to access the message queue created by the KDC, hence corrupting our use case.

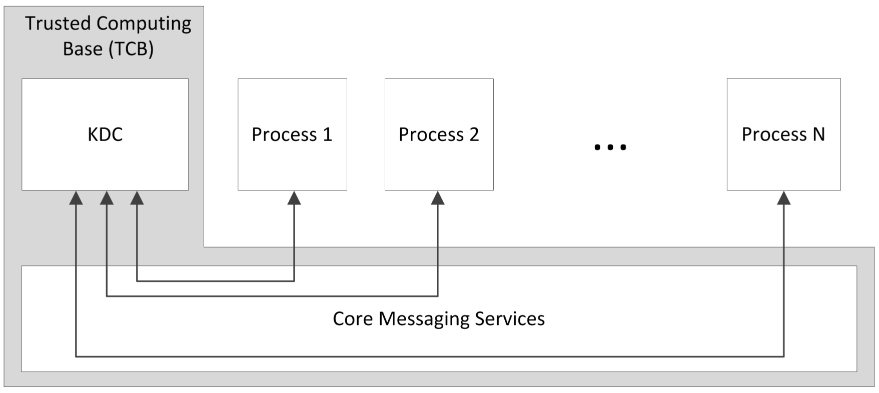


Figure 5.6: Missing authorization mechanism.

5.4.5 CASE 4: Process tracing and debugging

5.4.5.1 *ptrace()* System Call

Almost all operating systems provide some sort of debugging. Debugging referes to allowing a process to examine the data and control of execution of another running process. This includes reading and writing arbitrary values from memory, register values, and signals. The kernel is the place where all the information of every process is stored (see chapter 5.3.3). Therefore debugging requires kernel support, because we need to tell the kernel that a certain process is a debugger and it is going to debug another process. In the case of Linux these operations and much more can be done by a single system call named *sys_ptrace*, which allows tracing and debugging within usermode. It provides the entry point to the code in the kernel that implements the *POSIX ptrace API*. User programs should call the *ptrace()* wrapper implemented in *glibc*. The prototype is shown below:

```
long ptrace(enum __ptrace_request request,
            pid_t pid,
            void *addr,
            void *data);
```

- `__ptrace_request request`:
  The operation ptrace shall perform.

- `pid_t pid`:
  The process ID to perform the operation on.

- `void *addr`:
  Designates the address in memory for which to read or write for certain ptrace operations; Ignored by some operations.

- `void *data`:
  Address for various data structures to be read/written to/from the process.

The *ptrace() system call API* defines different operations like:

- *PTRACE_TRACEME*
- *PTRACE_ATTACH*
- *PTRACE_DETACH*
- *PTRACE_GETREGS*
- *PTRACE_SETREGS*
- *PTRACE_PEEKDATA*
- *PTRACE_POKEDATA*
- etc.

The *ptrace()* system call establishes a connection between the tracing process (parent process) and the traced process (child process). There are two ways to achieve this. Either the parent literally forks the child process (*PTRACE_TRACEME*), or it attaches to the running child process during execution (*PTRACE_ATTACH*). After connecting to the child process, we can read (*PTRACE_PEEKDATA*) and write (*PTRACE_POKEDATA*) its memory, which is actually done by accessing the process' kernel structures *mm_struct* and *vm_area_struct* (see chapter 5.3.3.2). We can also read and write one process' registers by using *PTRACE_GETREGS* and *PTRACE_SETREGS*. Another interesting feature may be the ability to trace the system calls done by a running process with *PTRACE_SYSCALL*.

5.4.5.2 Tracing the KDC

According to our use case we are running the KDC and 2 processes that are allowed to request keys from the KDC. Besides those two processes we now will execute another process that will trace the KDC's heap address space. Note that a process may be traced by only one process at a time and only by a PID owned by the same or by the root user. For tracing another process there are 2 possibilites. Either we could use the *GNU debugger (GDB)*, that actually utilizes *ptrace* for reading memory frames. Here the KDC should have been compiled with the debugging switch (*gcc -g*). Or we could write our own code using the *ptrace glibc* command, what actually was our choice (see listing 5.1).

```c
int main(int argc, char **argv){
    pid_t pid;
    unsigned long addr1, addr2;
    long buf;

    ...
    /* command line parsing */
    ...

    if(ptrace(PTRACE_ATTACH, pid, NULL, NULL)){
        perror("PTRACE_ATTACH");
        return(1);
    }

    waitpid(pid, NULL, 0);

    for(; addr1 < addr2; addr1 += sizeof(unsigned long)){
        errno = 0;
        if(((buf = ptrace(PTRACE_PEEKDATA, pid, (void *)addr1, NULL))== -1)
             && errno){
            perror("PTRACE_PEEKDATA");
            if(ptrace(PTRACE_DETACH, pid, NULL, NULL)){
                perror("PTRACE_DETACH");
            }
            return(1);
        }
        printf("addr: %08lX | data: %08lX\n",addr1, buf);
    }
    if(ptrace(PTRACE_DETACH, pid, NULL, NULL)){
        perror("PTRACE_DETACH");
        return(1);
    }
    return(0);
}
```

Listing 5.1: tracemem.c

Tracemem awaits 3 parameters: the PID of the process we want to trace, the start address *addr1*, and the end address *addr2* of the address space we want to read the data from:

usage: *tracemem <pid> <address1> <address2>*

After attaching to a process with the corresponding PID, *tracemem* reads 4 bytes at a time from the specified address space. In the following we will examine the steps that are actually necessary to carry out a memory dump under the usage of *tracemem*.

1. Start the KDC and find out PID (e.g. *ps -a* or debug output of KDC).

   ```
   user@desktop:> ./kdc
   PID: 5797
   ==========
   KEY_DB:
       key_id: 1; value: 00112233445566778899AABBCCDDEEFF
   ==========
   ```

2. Find the heap address space with *cat /proc/PID/maps*.

   ```
   user@desktop:> cat /proc/5797/maps
   ...
   09cd1000-09cf2000 rw-p 00000000 00:00 0          [heap]
   ...
   b7786000-b7787000 r-xp 00000000 00:00 0          [vdso]
   ...
   bffad000-bffce000 rw-p 00000000 00:00 0          [stack]
   ```

3. Before tracing we need to stop the KDC by sending the SIGSTOP signal.

   ```
   user@desktop:> kill -s SIGSTOP 5797
   ```

4. Trace the address space of the heap taken from step 2 and redirect output to a file.

   ```
   user@desktop:> sudo ./tracemem 5797 09cd1000 09cf2000 > heapdump
   ```

5. Examine the tracefile for heap data, where in the end we find our pre-stored key.

   ```
   user@desktop:> vi heapdump
       Memory Dump
   ======================
   process id: 5797
   start address: 9CD1000
   end address: 9CF2000
   ======================

   addr: 09CD1000 | data: 00000000
   addr: 09CD1004 | data: 00000021
   addr: 09CD1008 | data: 00000000
   addr: 09CD100C | data: 00000001
   addr: 09CD1010 | data: 00112233
   addr: 09CD1014 | data: 44556677
   addr: 09CD1018 | data: 8899AABB
   addr: 09CD101C | data: CCDDEEFF
   addr: 09CD1020 | data: 00000000
   ...
   ```

5.4.5.3 Security Concerns: *ptrace*

It is clear that *ptrace* introduces a serious security issue since malicious software could use it to read the memory of another process. If allowed normally, SSH session hijacking and even arbitrary code injection is fully possible. You may find examples on the net or in applicable literature [PO10]. Since Ubuntu 10.10 for example, *ptrace* is only allowed directly from a parent to a child process, or as the root user. In newer kernels also the usage of the pseudo-filesystems like the */proc* filesystem is further restricted.

5.4.6 CASE 5: Loadable Kernel Modules and Device Drivers

5.4.6.1 Overview

Loadable kernel modules (LKMs) offer the opportunity to extend the functionality of the Linux kernel. Kernel developers can write their own kernel code and after compilation add it to the kernel at runtime without having to rebuild the kernel. The key feature of LKMs is that they are dynamically loaded when their functionality is required. Afterwards they get unloaded, which actually saves kernel memory. LKMs are used for various things like device drivers and filesystems (e.g. the */proc* filesystem). Even if LKMs are loaded dynamically, they in fact are kernel code and operate in kernel-space with all its privileges. This also includes access to the memory area of user-space processes. In the following we will develop a kernel module that prints the heap of our KDC user space process in order to explore the key information. Loadable kernel modules come in handy to show that we actually can do anything in the kernel, and that we can do this quite fast withouth recompiling the whole kernel. Of course this introduces some serious security issues (see chapter 5.4.6.3).

5.4.6.2 Module

In chapter 5.3.3.2 we already heard that the Linux kernel manages a processes' memory by maintaining a *mm_struct* struct that lies within the *task* struct. So our goal is to write a kernel module that is able to access the *mm_struct*, moreover the heap of the running KDC. Since running in privileged kernel mode this of course is possbile, hence we have to take some considerations into account.

When the Linux kernel starts a process, the code that loads the executable fills in the kernel data structures stored in *task*. The process code is devided into memory segments like *BSS, Text, Data, Stack,* and the *Heap* segment. The start and end virtual user space addresses for the heap segment are stored in *mm_struct→start_brk* and *mm_struct→brk* of the corresponding task. We will have to scan this user space area to find our dynamically allocated debug key of the KDC.

In chapter 5.3.3.1 we already heard about memory management and virtual memory in Linux as a basic mechanism for protecting our system. Through the virtual memory approach and its implied separation, we end up with a separate memory segment for the kernel, and one for each of the processes. So a user space pointer by itself does not reference a unique location in memory, but only a location in a memory segment that belongs to a specific process. Therefore the kernel cannot directly dereference user-space pointers, including our heap virtual user addresses in *mm_struct*. In more detail this has the following reasons:

- A user-space pointer may be invalid in kernel-space because there may be no mapping for that address, or it could point to some other, random data. This situation depends on the architecture running the kernel and on how the kernel was configured.
- Compared to kernel-space memory, that is in RAM all the time, user-space memory is paged and may be swapped out. So even if the user-space pointer does mean the same thing in the kernel-space, it may happen that the corresponding page is not present in RAM. So referencing the user-space memory directly could generate a page fault, which could in the worst case crash the whole system.
- Most of the time a kernel will need to dereference pointers that come from a user program issuing a system call. The user program could be buggy or malicious, therefore it is no good idea to blindly dereference a user-supplied pointer. This could provide the program to access or overwrite memory anywhere in the system.

More details about how Linux actually manages memory and how addressing in the kernel works you should look up on your own. With the considerations about dereferencing pointers in the kernel in mind, we now will go on to our kernel module code that traces the KDC's heap.

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/highmem.h>
#include <linux/vmalloc.h>
#include <linux/slab.h>
#include <asm/page.h>
#include <asm/uaccess.h>

static int pid_mem = 1;

static int print_mem(struct task_struct *task){
    struct mm_struct *mm;
    struct page **pages;
    unsigned long *k_page;
    unsigned long *k_page_end;
    unsigned long heap_nr_pages;
    int res, i;

    /* get mm_struct of corresponding task */
    task_lock(task);
    mm = task->mm;
    if (mm)
        atomic_inc(&mm->mm_users);
    task_unlock(task);
    if (!mm)
        return 0;

    /* print heap status to kernel ring buffer */
    heap_nr_pages = (mm->brk - mm->start_brk) / PAGE_SIZE;
    printk("\nMemory Dump\n");
    printk("======================\n");
    printk("heap address range: %08lX -- %08lX;\n", mm->start_brk, mm->brk);
    printk("%lu addresses in %lu pages;\n", mm->brk - mm->start_brk, heap_nr_pages);
    printk("======================\n\n");

    /* allocate array of pages to store physical page information */
    if ((pages = kmalloc(heap_nr_pages * sizeof(*pages), GFP_KERNEL)) == NULL)
        return -ENOMEM;

    /* sem on mm_struct; get_user_pages for faulting in physical pages and storing
       addresses in "pages" */
    down_read(&mm->mmap_sem);
    res = get_user_pages(task, mm, mm->start_brk, heap_nr_pages, 1, 0, pages, NULL);
    up_read(&mm->mmap_sem);

    /* map every physical page to kernel memory, to address heap data */
    for(i = 0; i < res; i++){
        k_page = (unsigned long *) kmap(pages[i]);
        k_page_end = (unsigned long *)(k_page + (PAGE_SIZE / sizeof(unsigned long)));

        while( k_page < k_page_end){
            if(*k_page != 0)
                printk("page: %d, k_addr: %p, data: %08lX\n", i, k_page, *k_page);
            k_page++ ;
        }
        kunmap(pages[i]);
    }
    kfree(pages);
    return 0;
}

static int mm_exp_load(void){
    struct task_struct *task;
    printk("\nGot the process id to look up as %d.\n", pid_mem);
    for_each_process(task) {
        if (task->pid == pid_mem) {
            printk("%s[%d]\n", task->comm, task->pid);
            print_mem(task);
        }
    }
    return 0;
}

static void mm_exp_unload(void){
    printk("\nPrint segment information module exiting.\n");
}

module_init(mm_exp_load);
module_exit(mm_exp_unload);
module_param(pid_mem, int, 0);

MODULE_AUTHOR ("Juergen Broder, juergen.broder@gmail.com");
MODULE_DESCRIPTION ("Print Process Pages");
MODULE_LICENSE("GPL");
```

Listing 5.2: procinfo.c

Our kernel module *procinfo* expects as paramater a Process ID that identifies the process we want to read the heap from. In our case this of course is the PID of the KDC. Then it searches for the corresponding task in the kernel's *task struct* and when found handles it to the *print_mem()* routine. There we access *mm_struct* to find out the KDC's heap address range which probably will span over more then one page. The number of heap pages is stored in *heap_nr_pages*. We heard before that user pages may not be present in memory when dereferencing and that there might be no mapping between the user space and kernel space pointers. Fortunately the kernel programming API offers many ways to work around this problem.

First of all we allocate an array of *page structs* for each page of the KDC heap pages. A *page* struct holds information about the physical memory pages including the physical start and end addresses of the page. Then we need to ensure that those pages are faulted in into the physical memory which is achieved by the *get_user_pages()* routine. The details you might look up on your own, but the important part here is that the specified pages will be loaded into memory and that the physical address information then gets stored in the *pages* array. After this we need to map the physical page addresses to kernel virtual addresses to actually access the corresponding data. This is achieved by the *kmap()* routine that awaits a *page struct* as parameter and returns a kernel virtual address. In this way we are able to iterate through the pages and print out data if found. On every iteration we unmap the old page and map the next one. When done the memory for *page struct* is freed. In the following we will describe the steps to carry out for reading the heap memory of the KDC using our *procinfo* module.

1. Start the KDC.

   ```
   user@desktop:> ./kdc
   PID: 5797
   ==========
   KEY_DB:
       key_id: 1; value: 00112233445566778899AABBCCDDEEFF
   ==========
   ```

2. Insert the *procinfo* kernel module with the KDC's PID.

   ```
   user@desktop:> sudo insmod procinfo.ko pid_mem=5797
   ```

3. Examine the kernel ring buffer.

```
user@desktop:> dmesg

[ 2412.199180] Got the process id to look up as 5797.
[ 2412.199206] kdc[5797]
[ 2412.199207]
[ 2412.199208] Memory Dump
[ 2412.199209] =====================
[ 2412.199210] heap address range: 09CD1000 -- 09CF2000;
[ 2412.199212] 135168 addresses in 33 pages;
[ 2412.199213] =====================
[ 2412.199213]
[ 2412.199228] page: 0, k_addr: ffc2c004, data: 00000021
[ 2412.199229] page: 0, k_addr: ffc2c00c, data: 00000001
[ 2412.199231] page: 0, k_addr: ffc2c010, data: 00112233
[ 2412.199232] page: 0, k_addr: ffc2c014, data: 44556677
[ 2412.199234] page: 0, k_addr: ffc2c018, data: 8899AABB
[ 2412.199235] page: 0, k_addr: ffc2c01c, data: CCDDEEFF
[ 2412.199237] page: 0, k_addr: ffc2c024, data: 00020FE1
```

5.4.6.3 Security Concerns: Loadable Kernel Modules

Since loadable kernel modules are in fact kernel code that runs under the highest privileges, it is clear that they are a serious security issue to the system and therefore are targeted by malicious users. Inserting a module requires super user privileges which one might have gained by exploiting a known vulnerability or retrieving the root password either by cracking, privilege escalation (see chapter 5.4.2.1), or social engineering. Then the user might compromise the whole system even at runtime, since kernel modules are loaded dynamically without the need to restart the system.

A very popular utilization of loadable kernel modules is the development of so called *rootkits*. Rootkits may be placed in user space or kernel space. A kernel rootkit in general is malicious software that hides the existence of certain processes or programs from the detection of normal software (e.g. antivirus software). It can also be used to enable continued privileged access to user programs. To achieve this goal kernel rootkits use their privileges to modify for example the detection software or the system call table to subvert kernel functionality and bypass security checks. Hence they are able to intercept or subvert the most trusted operating system operations. In this situation, no part of the system can be trusted. Kernel rootkits are more difficult to write then user rootkits because of the operating system's complexity.

5.5 PikeOS

5.5.1 PikeOS System Architecture

PikeOS is a microkernel-based (see chapter 3.5) real-time operating system from SYSGO AG [SYS12] that is used to employ safety and security critical embedded systems. Since we are interested in security, we will only have a look at security relevant features. The primary security mechanism is based on a *resource partitioning* approach that allows to host multiple virtual machines (see chapter 3.4) on top of PikeOS. Through this partitioned environment, multiple operating systems and applications with different criticality levels may run in a secure environment on a single machine without interfering with each other. Then entire operating systems along with their applications can be hosted within a virtual machine. Since PikeOS uses paravirtualization (see chapter 3.4.4.2), operating systems need to be ported and adapted to run in one of the VMs, whereas applications can run unmodified. PikeOS is also able to host different APIs and run-time-environments (RTEs, see figure 5.7). Each VM has its own, separate set of resources making programs hosted in one VM independet from those hosted by another, thereby enforcing ***isolation***. Besides hosted native applications, RTEs and OSs, also drivers (see chapter 5.4.6) and stacks reside in separate address spaces with pre-defined I/O access.

Besides this spatial separation described above, PikeOS also provides a separation of temporal resources amongst its client OSes and applications, with a patented scheduling method. With this approach it is possible to virtualize hard real-time systems while still retaining their timing properties. However, temporal separation will not be of our concerns, since we want to spatially separate the KDC and our trusted programs from the untrusted ones, therefore using a standard scheduling mechanism with equal time slots for every partition. In PikeOS two basic terms are concerned with the execution and scheduling of applications:

- **Thread**: A thread is the basic PikeOS scheduling entity. It has access to data and stack memory and is associated with a time partition, a task and the task's resource partition.
- **Task**: Each task has a seperate address space where all the threads assigned to this task share this memory.

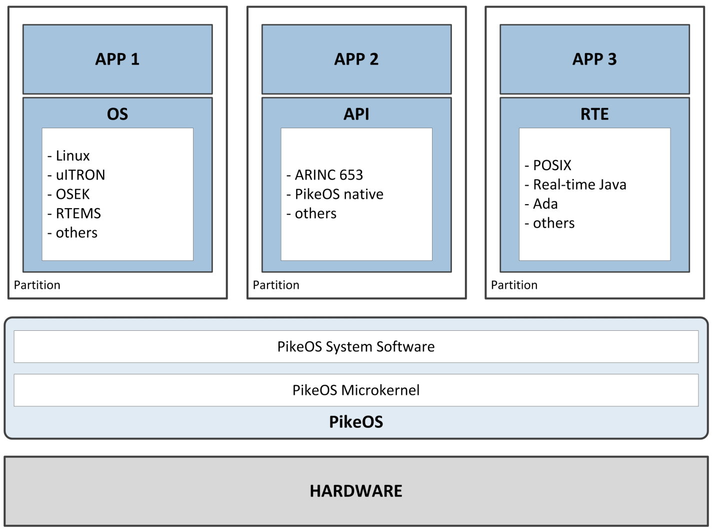


Figure 5.7: The PikeOS System Architecture

5.5.2 PikeOS System Software

The *PikeOS System Software (PSSW)* is built on top of the PikeOS microkernel. Many of the services normally found inside a monolithic kernel, like file system access, system configuration, application process control, or starting applications, are found in the PSSW that is actually running in user mode. In addition and less common to a monolithic kernel, it provides the services to implement partitions, like partition control and inter partition communication. The PSSW also assigns the temporal and spatial resources statically to the individual VMs, hence is responsible for establishing the system configuration at the system's start. The parameters for all PSSW services are stored in the so called *Virtual Machine Initialization Table (VMIT)* which is loaded by the PSSW at boot time. After the initial configuration, the PSSW establishes the various partitions. This is done by assigning the resources and afterwards starting a partition deamon thread for each partition that is responsible for the interaction between the partitions and the PSSW module. User applications request services of the PSSW module by means of the PSSW library. From the initial configuration point on, now per default all occuring errors only affect the actual partition. Afterwards all applications processes in the partitions are loaded and started. The PSSW library is linked to the user applications and hides the PikeOS kernel's inter process communication protocol that is used to communicate with the PSSW module from the user application.

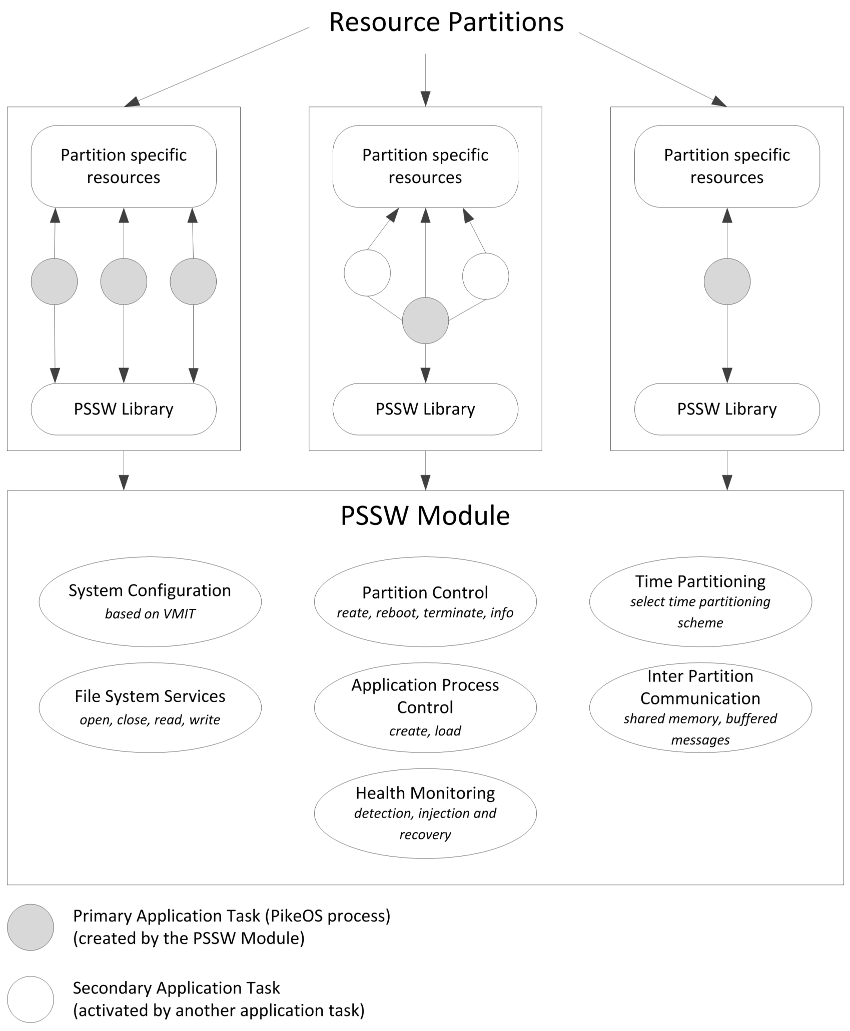

Resource Partitions
Partition specific resources
Partition specific resources
Partition specific resources
PSSW Library
PSSW Library
PSSW Library
PSSW Module
System Configuration
based on VMIT
Partition Control
reate, reboot, terminate, info
Time Partitioning
select time partitioning scheme
File System Services
open, close, read, write
Application Process Control
create, load
Inter Partition Communication
shared memory, buffered messages
Health Monitoring
detection, injection and recovery
Primary Application Task (PikeOS process)
(created by the PSSW Module)
Secondary Application Task
(activated by another application task)

Figure 5.8: The PikeOS System Software Architecture [SYS12]

5.5.2.1 Resource Partitions

Resource partitions can be imagined as a container for user applications that provide *protection domains* (see chapter 3.2) between those applications. We already heard that the resource partitions are created by the PSSW module. They have a statically defined set of resources and privileges, that are loaded from the VMIT. The partitions and their corresponding configurations cannot be modified at system runtime. Every partition has a set of attributes including the *Partition Identification, Privilege Level, Maximum Priority, Associated Time Partition, Maximum Number of PikeOS Child Tasks, File Access Privileges*, and so on. The privilege level determines whether a partition is a *system* or *user* partition. Applications running in a system partition have permission to perform operations that may affect other partitions, therefore those partitions must be *trusted*. Examples for application running in a system partition would be a health monitor application or a service provider.

Every application running in a partition can use a set of resource entities that need to be configured for each partition, including *Memory and I/O, File Services, Associated Application Processes* and *Partition Communication Ports*. A partition always accesses the same memory that has been allocated to it by the partition configuration. This memory will never be returned to the system, even if the partition shuts down. After restarting, it again will access the same memory location. This actually ensures that applications behave predictively, once the resources are available they won't be exhausted in a subsequent run. In addition it is not possible that information gets accidentally or maliciously exchanged between partitions through a reassignment of physical memory pages.

5.5.2.2 Application Process Management

All the applications that are started by the PSSW module are called *processes*. For the kernel these processes are simply PikeOS tasks (see chapter 5.5.3.3). The PSSW assigns properties to these tasks, hence adding additional semantics to them. Processes then access the PSSW services through the PSSW library that is linked to every partition. The PSSW performs several steps for each process that is defined in a partition, including to allocate a PikeOS task from the partition's allocated task memory pool, creating all mappings in the process address space, map or copy files into the process address space, donate configured number of PikeOS child tasks to the process, and so on. These steps are carried out by the a partition daemon in the PSSW module that is started for every partition. In a PikeOS resource partition one or more processes can be started. They have a different virtual address space, but share the other resources assigned to the partition.

Every process has different attributes including a process name and ID, a command line, virtual address layout describing memory mappings and assigned I/O resources to the partition, list of files to be loaded into the address space of the process, maximium controlled priority (MCP) which is the maximum scheduling priority for the process, maximum number of child tasks and maximum number of threads that can be created by the process.

5.5.2.3 Partition Communication Ports

Communication between resource partitions is based on messages that are sent from one *partition communication port (PCP)* to another over *communication channels*. A communication channel is a link between PCPs where data flows from a *source PCP* to a *destination PCP*. Thereby the source and destination PCPS may be located within the same partition, within two different partitions, or within one partition and a system extension. PCPs are configured in the VMIT and are a partition specific object. They are created by the PSSW module on system boot. Once PCPs have been created, they always exist throughout the system's runtime and will never be deleted. PikeOS differs between two kinds of ports: *queuing ports* and *sampling ports*.

- ***Queuing ports***, quite like POSIX message queues, work on a FIFO strategy and buffer messages up to a maximum number of queued messages. The messages written to a queuing port are buffered within the PSSW module, ensuring that it cannot be corrupted by any user application.

- On the other hand ***sampling ports*** buffer only one message but compare this message at a given refresh rate and report the validity of the stored message. Both types of PCPs share the *port name, port direction*, and *maximum message size* properties. However queuing ports define a maximum number of queued messages, whereas sampling ports define a refresh rate for sampling incoming messages.

5.5.2.4 Port Communication Channels

Port communication channels (PCCs) can only connect together ports of the same type, where each channel has exactly two endpoints: one *source* and one *destination port*. A queuing port, whether source or destination and a destination sampling port can only belong to one channel. On the other hand a source sampling port can be referenced in an arbitrary number of channels, which allows a configuration where one source sampling port broadcasts messages to several destination sampling ports. Ports that are connected over channels must not only be of the same type, but must also be compatible, meaning that their properties (except for port direction) must match.

The properties of a PCC are it's name, source, and destination port and are specified in the VMIT. Once again like PCPs, all PCCs are created by the PSSW module at boot time. A PCC will never be deleted and will exist throughout the runtime of the system. Applications use communication channels implicitly by accessing the PCPs, there is no channel API.

5.5.3 PikeOS Microkernel

The *PikeOS microkernel* is compliant to the MILS architecture (see chapter 5.7.1). It provides a core set of services like I/O and interprocess communication and is responsible for supervising every access to the hardware. Since it only contains of about 5000 lines of code it is simple enough for a formal analysis of its properties. The functionality of the microkernel is based on four concepts and mechanisms, the *resource partitions, time partitions, tasks* and *threads*.

5.5.3.1 Resource Partitions

We already heard in chapter 5.5.2 that the PSSW module is responsible for managing resource partitions. Seen from the kernel side a resource partition is a set of PikeOS *tasks*, which share a bounded set of *kernel resources*. Kernel resources are all those that the kernel needs to offer its services. When concerned with resource partitions the managed resource is *kernel memory*. This memory is used for all allocations in the kernel, like task descriptors, thread control blocks, and memory management. The kernel creates all resource partitions at boot time whereby all, except partition 0, in the beginning are empty. Partition 0 gets initialized automatically and starts an initial task. Once created a resource partitions can't be deleted, so the resources assigned to the partitions can't be retrieved again. Only the tasks of a resource partition may be deleted.

5.5.3.2 Time Partitions

We heard that a thread is the basic scheduling entity of PikeOS and that each thread belongs to a task. PikeOS handles thread scheduling by means of *time partitions*. Every thread gets assigned to a specific time partition, whereby threads from different tasks may reside within this partition. The kernel schedules over time partitions by activating one at a time. For a thread to be scheduled for execution it must be in the ready state, must have the highest priority of all the ready threads, and the thread's associated time partition must be active. The time partition mechanism is especially interesting for hosting native operating systems (Linux) besides real-time applications that need to meet hard timing deadlines. We won't be concerned with time partitions and will use a basic scheduling mechanism with equal time slots for every time partition.

5.5.3.3 Tasks

PikeOS defines *tasks* as an address space and a set of schedulable entities (threads) bound to the task. In the following we will describe important task attributes the kernel uses for defining a task.

- **Task ID**: Task identifier.
- **Task name**: String defining the task's name.
- **Mappings**: Mappings between virtual and physical addresses.
- **Abilities**: Define which kernel services (system calls) a task can utilize (see chapter 5.5.3.7). Will be specified during task activiation.
- **Resource Partition**: The task's associated resource partition.
- **Communication Rights**: Define if a task is allowed to communicate with another task (see chapter 5.5.3.4).
- **Maximum number of threads**: Maximum number of threads per task. This attribute is usually used to limit kernel resource consumption.

The special task ID 0 is reserved for the PikeOS kernel task and cannot be accessed from any user program. The task with ID 1 is the system's initial task also called *root task* and is the first task running in user space. It is responsible for starting further tasks and donating requested resources to them. So all tasks are child tasks of the root task, but are also allowed to create further child tasks.

5.5.3.4 Threads

We already heard that threads are the schedulable entities of a task. For each task the kernel allows a maximum of *P4_NUM_THREAD* threads. Some important thread attributes are:

- **TaskID:** Defines the task to which a thread belongs. Will never be changed.
- **Thread UID:** Used to uniquely identify threads throughout the system and used for example during IPC.
- **Execution Priority:** The thread's priority for scheduling. Can never be higher then the task specific MCP.
- **Time Partition:** The time partition a thread belongs to.
- **Event Mask:** Restricts the range of threads that are allowed to send signals to a thread.
- **IPC Mask:** Restricts the range of threads that are allowed to send IPC messages to a thread.

5.5.3.5 Communication

The kernel also provides two communication mechanisms to allow threads to exchange data and synchronize with each other:

- **Interprocess communication (IPC)**: Used for transferring unbuffered messages between two threads *synchronously*.
- **Event communication**: Threads *signal* events to another thread *asynchronously*.

Those two mechanisms can be compared to message queues and signals in Linux we already dealt with in chapter 5.3.3.3 and when implementing the KDT (see chapter 4). In PikeOS however, transferring a message always consists of a sender and receiver part. Receiving threads always need to specify the thread(s) they want to receive messages from. Sender threads on the other hand always must specify a single *Unique ID (UID)* to designate the target receiver thread. UIDs are the standard mechanism of PikeOS to address threads and are unique in the whole system. They are composed of the time partition number the thread is executing in, the resource partition number and task ID the thread is assigned to, and the thread ID by itself. Besides specifying the receiver thread, a sender thread must possess the *right* to communicate with the receiver.

In PikeOS communication between threads is controlled by *communication rights* offering an additional layer of security. Although communication takes place between different threads, communication rights are a task attribute (see chapter 5.5.3.3). Communication rights take effect when threads want to communicate with threads in another task. Therefore all threads belonging to one task share the same communication rights, hence are allowed to communicate with each other. Every task will be created with a set of standard rights which cannot be changed during the lifetime of a task. These standard rights include that a task by default may communicate with its parent and it's direct children. Beyond standard rights a parent task may *grant* it's child task additional rights, but only if the parent itself possesses the right for granting. Through these restrictions, communication between tasks can only take place by the presence of a common ancestor. Rights are statically defined per task and cannot be changed during the lifetime of a task.

5.5.3.6 Memory Management

Like Linux also PikeOS implements virtual memory where all user code operates on virtual addresses that are translated to physical addresses by the MMU of the CPU. Every task is assigned its own virtual address space in the range from *P4_MEM_USER_BASE* to *P4_MEM_USER_END*. Addresses outside this range cannot be accessed by user code and are only accessible from the PikeOS kernel. Hence the PikeOS memory management approach is the same as in Linux (see chapter 5.3.3).

5.5.3.7 Abilities

Another important security feature of PikeOS are *task abilities*, that define if a task is allowed to use certain kernel services, which a task requests by issuing system calls. Abilities get associated to tasks and will be stored in the task attributes (see chapter 5.5.3.3). The kernel then uses the abilities to verify if a task's thread has sufficient permissions for the requested kernel service. The thread is allowed to use kernel services if the ability is *enabled*, hence not allowed if it is *disabled*. Abilities are define per task, therefore all threads in the task share the same set of abilities. Once again, abilities are assigned at task activation time and can not be changed during the task's lifetime. When a parent task creates a child task, the child inherits all abilities from the parent. The abilitiy set of the child tasks may be further restricted by the parent, but it is not allowed to add new abilities that the parent itself has not enabled. Later in the setup of our use case in PikeOS and the case comparison we will have a look at abilities that may be important to break the memory acquisition cases.

5.6 Key Management in PikeOS

5.6.1 Porting the KDT

After looking at the theoretical background we will port the Key Distribution Tool to PikeOS according to our use case. PikeOS is shipped with the *CODEO IDE* that allows project development over a graphical user interface. Software development in PikeOS is devided into *Application Projects* and *Integration Projects*. Application projects can be thought of as writing native PikeOS applications that use the APIs provided by the PSSW. Integration projects are used to plan and specify the partition configuration, bootstrap strategy, and also for creating the ROM image that includes the applications developed in the application projects. The bootable ROM image makes up the whole PikeOS system and may be downloaded to an embedded device. We will build a bootable ROM image for the *PHYTEC phyCORE-i.MX35* development board (see chapter 5.6.3).

5.6.2 Application Development and the POSIX Personality

We heard that PikeOS is able to host different run-time-environments including a POSIX RTE. Therefore we start separate application projects for the KDC and each of the trusted user programs. By using the POSIX API only little changes to the structure of the code were necessary, contrarely it even became smaller because of the architectural benefits PikeOS offers (see chapter 5.7.3). The different hosted RTEs in PikeOS are called personalities. The POSIX personality implements the *PSE51 profile* of the *IEEE Std 1003.13-1998* with some additional realtime extensions included. A POSIX application consists of one PikeOS task in which several PikeOS threads are running.

When compiling applications in the POSIX personality API, the *libc.a* and *libm.a* libraries will be linked to the application files. The *libc.a* library contains the standard C-language API functions, the POSIX thread API and the "operating system run-time". On the other hand *libm.a* contains the Math-Library API functions. Besides those two basic libraries we also need the *libtomcrypt.a* and *tomsfastmath.a* libraries for our cryptographic processings. In order to make them available for the *phyCORE board* and it's *ARM architecture* we need to cross compile them by using the *cross development kit (CDK)* of PikeOS.

5.6.3 The Hardware

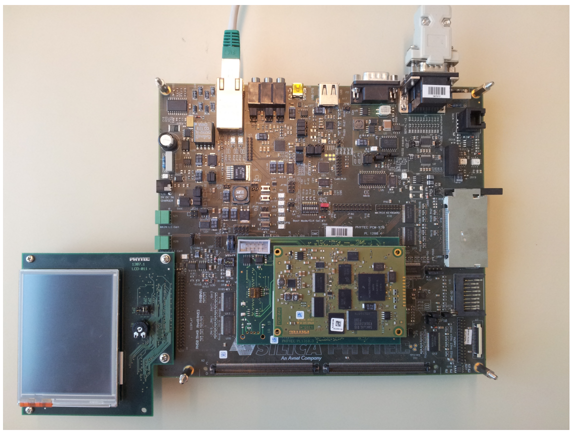


Figure 5.9: phyCORE-i.MX35 Rapid Development Kit

CPU	
Processor	Freescale i.MX35x
Architecture	ARM-1136
Frequency (max)	532 MHz
MMU	yes
MEMORY	
On-Chip	16 KB L1, 128 KB L2, 128 KB SRAM
SRAM	-
DRAM	128 MB DDR2
NAND Flash	1 GB
NOR Flash	32 MB
EEPROM	4 KB

Table 5.1: Hardware Features

5.6.4 Block Diagram

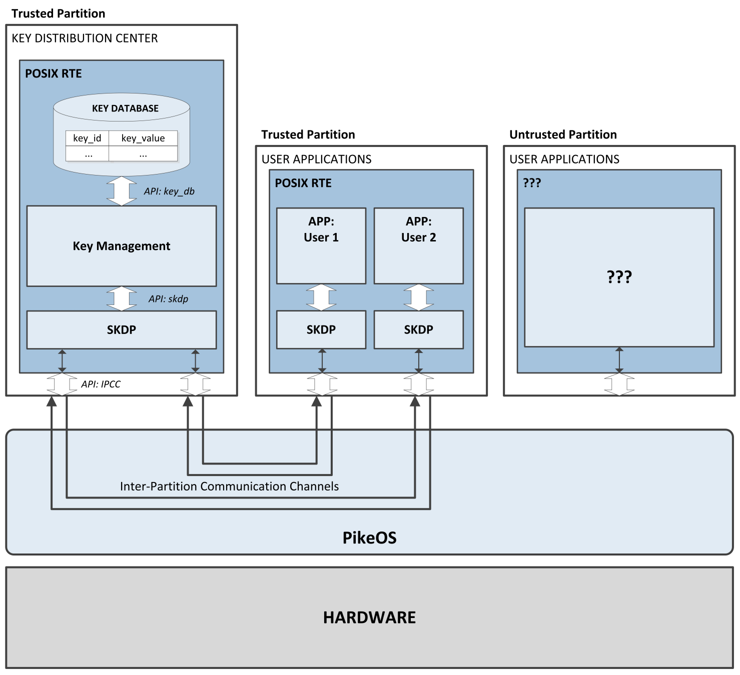


Figure 5.10: Key Management Tool in PikeOS

5.6.5 Resource Partitions

We heard that besides developing applications we also need to setup the PikeOS system configuration in an integration project. One of the first tasks by doing so, is to define and configure the resource partitions that enforce protection domains between applications. In the next step the applications developed in the application projects need to be assigned to those partitions. According to our use case we defined 3 resource partitions that are hosting applications. Some of the properties are listed in the tables below. An overview is given in figure 5.10 in chapter 5.6.4.

5.6.5.1 KDC Partition

Contains only the Key Distribution Center (KDC) to be sure that no other application will interfere with our key database.

Property	**Value**	**Description**
PartitionName	*kdc-part*	the partition's name
PartitionIdentifier	*1*	unique partition identifier
TimePartitionID	*1*	associated time partition (ch. 5.6.6)
SystemPartition	*false*	privilege level: user partition (ch. 5.6.8)
MaxChildTasks	*1*	restrict partition to only run 1 task
QueuingPortList	*kdc_OUT_1* *kdc_IN_1* *kdc_OUT_2* *kdc_IN_2*	list of all queuing ports (ch. 5.6.7)
ProcessList	*kdc*	runs the kdc process only

5.6.5.2 Trusted User Partition

Contains the 2 user applications that are allowed to request keys.

Property	**Value**	**Description**
PartitionName	*trusted-users*	the partition's name
PartitionIdentifier	*2*	unique partition identifier
TimePartitionID	*2*	associated time partition
SystemPartition	*false*	privilege level: user partition
MaxChildTasks	*2*	restrict partition to run 2 tasks
QueuingPortList	*user_OUT_1* *user_IN_1* *user_OUT_2* *user_IN_2*	list of all queuing ports
ProcessList	*user-1* *user-2*	runs the 2 trusted user applications

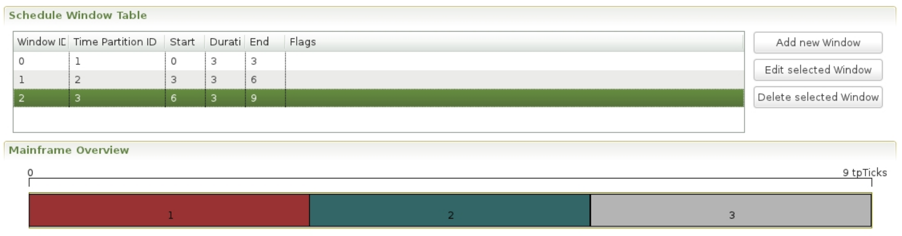
5.6.5.3 Untrusted User Partition

This partition is empty but would contain all the other, untrusted software that actually is not allowed to request keys.

Property	**Value**	**Description**
PartitionName	*untrusted-users*	the partition's name
PartitionIdentifier	*3*	unique partition identifier
TimePartitionID	*3*	associated time partition
SystemPartition	*false*	privilege level: user partition
MaxChildTasks	*X*	arbitrary
QueuingPortList	*X*	arbitrary
ProcessList	*X*	arbitrary

5.6.6 Time Partitions

As stated in one of the earlier chapters we will only use a simple time partitioning setup. Therefore each resource partition gets associated it's own time partition, which is defined in the *TimePartitionID* property. In addition all time partitions will have the same amount of CPU time, hence equal sized time slots (see figure 5.11).



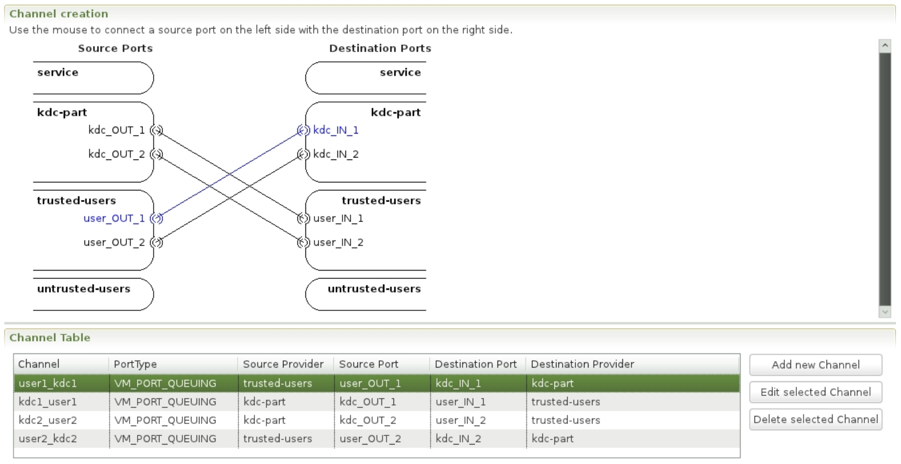
Schedule Window Table

Window ID	Time Partition ID	Start	Durati	End	Flags
0	1	0	3	3	
1	2	3	3	6	
2	3	6	3	9	

Figure 5.11: Time Partitioning Setup Key Distribution Tool

5.6.7 Communication Channels

When placing applications in their resource partitions they are completely separated from applications in other partitions. Therefore we need to define interpartition communication channels between partitions to enable communication beyond partitions. Since communication channels are uni-directional stream buffers we need to define at least two channels between two partitions. The one to let user tasks send requests to the KDC and the other to let the KDC send key responses back to the users. The combination of both ICCs can be seen as a bi-directional virtual communication channel. In our configuration we either went a step further and did not just set up channels between partitions, but defined such a bi-directional communication channel between every user application and the KDC. This has some serious benefits we will be talking about in chapter 5.7.3. The setup can be seen figure 5.12 and in the block diagram 5.10 in chapter 5.6.4.



Use the mouse to connect a source port on the left side with the destination port on the right side.

Channel	PortType	Source Provider	Source Port	Destination Port	Destination Provider
user1_kdc1	VM_PORT_QUEUING	trusted-users	user_OUT_1	kdc_IN_1	kdc-part
kdc1_user1	VM_PORT_QUEUING	kdc-part	kdc_OUT_1	user_IN_1	trusted-users
kdc2_user2	VM_PORT_QUEUING	kdc-part	kdc_OUT_2	user_IN_2	trusted-users
user2_kdc2	VM_PORT_QUEUING	trusted-users	user_OUT_2	kdc_IN_2	kdc-part

Figure 5.12: Channel Configuration

5.6.8 Privileges

All privileges in PikeOS are mangaged by the PikeOS kernel at task level. Those privilege levels are devided into:

- Abilities (see chapter 5.5.3.7)
- Communication rights (see chapter 5.5.3.4)
- Interrupt attachment rights (not of our concern)

Resource partitions can be configured as user or system partitions (see chapter 5.5.2.1), also referred to as the partition's privilege type, which defines a set of *abilities* that allow applications to access the PikeOS kernel. According to this privilege type, the PSSW as parent of all user applications statically installs the kernel privileges for all application processes that are running in the partitions. As opposed to user partitions, system partitions have an extended set of kernel privileges in order to execute PSSW services that also may affect other partitions or even the entire system. The privilege type is specified in the VMIT configuration by the *SystemPartition* attribute. In our use case setup, all resource partitions are defined as user partitions, thereby setting the *SystemPartion* attribute to *false* (see tables in chapter 5.6.5). When applications request services, the PSSW dynamically verifies the permissions defined in the VMIT configuration. Table 5.2 summarizes the abilities available to user- and system partitions and for completeness system extensions. System extensions provide a way to enhance certain aspects of the PSSW and its API and may be implemented as file- or port providers. We can see that user partitions have less privileges than system partitions. User partitions retain tracing abilities (*P4_AB_TRACE*), probably breaking our use case. This ability may be further restricted (see chapter 5.7.4). System extensions on the other hand have full access. Communication rights have already been discussed in chapter 5.5.3.4 and will again be discussed in chapter 5.7.2. Handling interrupt rights have no relevance for our evaluations.

Ability	**User Partition**	**System Partition**	**System Extension**
P4_AB_ANY_P4_SYSCALL	x	x	x
P4_AB_TIMEPART_CHANGE		x	x
P4_AB_RESPART_CHANGE			x
P4_AB_MEM_CREATE		x	x
P4_AB_KMEM_HANDLING			x
P4_AB_TRACE	x	x	x
P4_AB_PSP_RESET			x
P4_AB_PSP_CONSOLE		x	x
P4_AB_TIMEPART_SETUP			x
P4_AB_TIMEPART_ENABLE_DISABLE			x
P4_AB_RESPART_SETUP			x
P4_AB_MONITOR		x	x

Table 5.2: Resource Partition Abilities

5.7 CASE Comparison

5.7.1 CASE 1: Privilege Escalation and Exploitation

The PikeOS real-time operating system was designed to meet the main requirements of the *Multiple Independent Levels of Security (MILS)* architecture, which is based on the work of Rushby of the early eighties [Rus81] [Rus84]. We already dealt with this topic in chapter 2.1.4 when talking about multiple levels of security. MILS is a high-assurance security architecture, that defines a *separation kernel* which devides the system's memory into partitions by using the hardware memory management unit. Through the separation it is possible to securely execute both trusted and untrusted code on the system, therefore MILS only allows a controlled information flow between different non-kernel partitions.

In traditional operating systems with a monolithic kernel, most services such as device drivers, file systems, network stacks, and so on, run in privileged (superuser) mode. The drawbacks of this approach have already been discussed in chapter 3.5. As a consequence, many of the services have been pulled out of the kernel and put into a broad user-mode layer referred to as *MILS middleware*. On the one side this results in a small separation microkernel providing very specific functionality, where the security policies that must be enforced at this level are relatively simple. The PikeOS separation microkernel for example provides basic core services such as scheduling, process communication and synchronization, context switches, and interrupt and processor exception handling. On the other side, the code now running in user mode is subject to the kernel's security policy enforcement, that is acting as a hypervisor since PikeOS implements paravirtualization (see chapter 3.4.4.2). To enforce the security policy the separation microkernel intercepts privileged machine instructions, first checks the rights of the caller against the system configuration, and then actually executes them on the hardware.

The other parts of PikeOS are implemented in the *PikeOS System Software (PSSW)*, where services including device drivers and filesystems can be removed, added, or extended. This contributes to the scalability and flexibility of PikeOS. Together with the PikeOS microkernel, the PSSW forms a minimal layer of globally trusted code, the Trusted Computing Base (TCB). The small TCB and the modular design allow PikeOS to be used in projects that need certification according to industrial standards like *IEC 61508, DO178B, EN 50128*, and *EN 62304*. Examples are the development of the *Airbus A400M loadmaster workstation* and the *Airbus A350 Integrated Modular Avionics* devices.

Finally, according to MILS user applications running on top of PikeOS are empowered to enforce their own security policy, instead on relying on generalized kernel security services. Every partition has it's own security policy. The MILS architecture also defines that partitions can communicate with each other through communication channels that have been configured when the system was genereated. User partitions under no circumstances are allowed to access the system's hardware directly. As opposed to monolithic kernels, application-level security policy enforcement now in MILS is effective, because the MILS separation kernel ***guarantees*** control of information flow and data ***isolation***. The first time this guarantee is made at an assurance (see chapter 2.1.3.3) level that was almost impossible to achieve with monolithic kernels.

Since MILS is an architecture concept, it tells nothing about the degree of security that it actually reaches. The measurement and evaluation of security is done by the *Common Criteria (CC) IEC 15408 Standard* [Cri12] we already heard about in chapter 2.1.3.3. The CC defines a framework in which computer system users specify their security requirements. Then vendors implement the system and finally testing laboratories evaluate the product to determine if it meets the defined security requirements. The standard defines

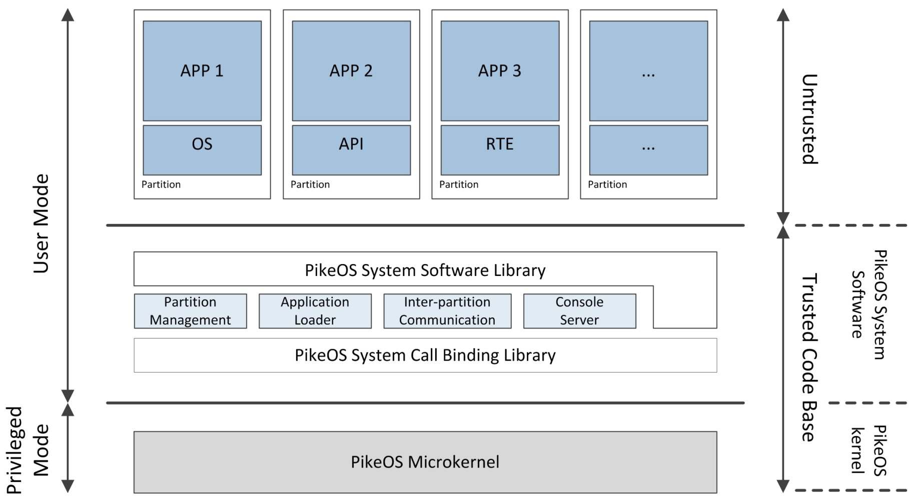


Figure 5.13: PikeOS Privilege Levels and Trusted Computing Base

seven levels of security, from from EAL 1 (lowest) to EAL 7 (highest). If a product is conformant to the Common Criteria Standard this will be claimed in a high-level document referred to as *Security Target (ST)*. Currently SYSGO is working on a security target, with a formal verification of the PikeOS microkernel on the way.

5.7.2 CASE 2: Sending Signals

In chapter 5.5.3.4 we saw that PikeOS offers IPC messages and events for communication between threads and that they need the right to communicate with each other. Sending signals in Linux applies to the event concept in PikeOS. Since we are using the POSIX personality API it actually is used in the same way by using the same system calls. In the following we will have a short look at the PikeOS communication API for granting communication rights. Afterwards we will have look some PikeOS internals that allow further restrictions on inter-process communication and sending signals, hence enforcing strong isolation between processes. Then we will compare the memory acquisition case in Linux (see chapter 5.4.3) to our use case setup in PikeOS. This will show that there is no way for an application to communicate with another if not permitted.

5.7.2.1 Granting Communication Rights

In PikeOS a task can grant one of its child task the right to communicate with other task if he has the appropriate rights for doing so. This can be done by calling one of the following two functions.

```
P4_e_t p4_comm_grant(P4_task_t dest,
                     const P4_task_bitmap_t *comm_map_p)
```

Calling this function grants task *dest* to communicate with other tasks.

- `P4_task_t dest`: Number of task receiving the communication rights. Must be a child task of the calling task.
- `const P4_task_bitmap_t *comm_map_p`: Pointer to bitmap in which every bit represents one task, the bit position corresponding to the task number.

```
P4_e_t p4_comm_link(P4_task_t dest,
                    P4_task_t which)
```

Calling this function grants task *dest* to communicate with task *which* and vice versa, hence establishing a communication channel between which and dest.

- `P4_task_t dest`: Number of task receiving communication rights with *which*.
- `P4_task_t which`: Number of task receiving communication rights with *dest*.

5.7.2.2 Thread Event Masks

In chapter 5.5.3.4 we saw that communication takes place between threads and consists of a sender and a receiver part. So besides communication rights, PikeOS offers a second opportunity to further restrict communication. This can be done by defining the *ipc_mask* and *ev_mask* attribute in the *P4_thread_create_str* struct.

```
struct P4_thread_create_str {
        const char * name;
        const P4_regs_t * context;
        P4_prio_t prio;
        P4_uint32_t tp_id;
        P4_uid_t shortexh;
        P4_uid_t fullexh;
        P4_uid_t ipc_mask;
        P4_uid_t ev_mask;
};
```

- `P4_uid_t ipc_mask`: Defines which thread(s) are allowed to send IPC messages to the newly created thread, by specifying a thread UID wildcard. IPC reception will be disabled by specifying *P4_UID_INVALID*.
- `P4_uid_t ev_mask`: Defines which thread(s) are allowed to send an event to the newly created thread, by specifying a thread UID wildcard. Event reception will be disabled by specifying *P4_UID_INVALID*.

5.7.2.3 Comparison

In our setup the three tasks (*kdc, user-1, user-2*) are created with standard communication rights. Hence sending signals among each other is impossible, since:

- The tasks are created independently and separated from each other, hence sharing no parent/children relationship.
- The tasks are not allowed to grant communication rights to children.
- The tasks consist of the initial thread only, not specifying any IPC or event mask.

Once again, in *Linux* access to system resources, including communication facilities like IPC message queues, is based on a user model. The drawbacks of this approach have been discussed extensively in chapter 5.4.2.1. Besides access control the only way to further restrict communication in Linux is blocking signals 5.3.3.3, which may also have certain disadvantages discussed in 5.4.3. Hence for a better separation and isolation between processes, Linux needs security enhancements (see chapter 5.3.4.2), or techniques like sandboxing (see chapter 3.3), which may suffer from race conditions.

In *PikeOS* however, communication between tasks and threads per default is not allowed, hence enforcing a strong isolation. So communication can concentrated on predefined inter-partition communication channels between partitions (see chapter 5.7.3). Since using the POSIX personality, blocking signals on threads through *sigprocmask()* would also be possible. But restricting event reception on a thread basis by event masks circumvents the drawbacks of blocking signals. Table 5.3 sums up communication relevant features in Linux and PikeOS.

	Linux	**PikeOS**
access control	per user (*coarse-grained*)	per task (*fine-grained*)
communication restriction	blocking signals	blocking signals
		communication rights
		IPC- and event-masks

Table 5.3: Communication comparison

5.7.3 CASE 3: Authorization and Interprocess Communication

5.7.3.1 Secure key distribution

In the previous case we saw that in our setup communication over IPC messages or signals is totally restricted. Therefore communication is limited to our predefined communication channels between partitions. This channels are actually used to exchange the messages for requesting keys from the KDC, hence implementing the SKDP. Since messages are buffered in the PSSW, they cannot be read by malicious users. Tasks outside our *trusted-users* partition also cannot request for keys, because there are no channels between other partitions (e.g. *untrusted-users* partition) and the KDC partition (see chapter 5.6.7). This setup also enforces clear separation between tasks with predefined communication flow, perfectly implementing our use case.

5.7.3.2 Comparison

As opposed to our Key Distribution Tool implementation in Linux, porting the KDT to PikeOS comes with some benefits:

- No need for an authentication mechanism in the KDC, because through predefined communication channels, the source or destination of messages can always be stated to one specific task (see chapter 5.4.4). Compared to Linux we now have a *connection-oriented* communication.
- Because of the message buffering in the PSSW, we need not encrypt communication by the Diffie-Hellman key exchange algorithm.
- Through this two improvements we also gain a smaller message struct. Because of the connection orientation, we need not pass source and destination information anymore (*src_qid, dest_qid*). Withouth Diffie-Hellman encryption we also don't need to pass the variable message size anymore (*data_size*). From now we are only transmitting fixed-size keys requested from the KDC and no variable size Diffie-Hellman public keys between requests (see listing 5.4 and 5.3).

```
typedef struct{
    int dest_qid;
    int src_qid;
    int data_size;
    int service;
    int key_id;
    unsigned char data[KEY_ARRAY_SIZE];
} messg_t;
```

Listing 5.3: messg_t in Linux

```
typedef struct{
    int service;
    int key_id;
    unsigned char data[KEY_ARRAY_SIZE];
}messg_t;
```

Listing 5.4: messg_t in PikeOS

5.7.4 CASE 4: Process tracing and debugging

5.7.4.1 *ptrace()*

In PikeOS and the POSIX personality tracing as specified by the *IEEE Std 1003.1-2008* standard is not supported. Therefore the kernel does not support the *ptrace()* system call, hence tracing the heap of the KDC from a user application as discussed in CASE 4 (see chapter 5.4.5) is not possible.

5.7.4.2 Debugging

The PikeOS POSIX personality and most embedded targets do not provide sufficient resources to run a fully-featured native debugger. Therefore the POSIX personality contains only a very small debugging agent also called *debug stub*. It is used in combination with a debugger (e.g. *GDB*) that is running on the host computer of system integrator. The the debugger and the debug stub then communicate over TCP/IP connection or a serial line, using GDB's standard *Remote Serial Protocol* for data exchange (*remote debugging*). Debugging for an application can be turned on or off by a switch in the application project. If enabled the GDB debug stub will be included in the application image.

In CASE 4 (see chapter 5.4.5) it was said that we could also use the GDB to read memory information of the KDC. In this native debugging approach the application is started under debugger control by attaching to the process over the *ptrace()* system call. In the embedded domain however, the programmer needs to call the *gdb_breakpoint()* function in the program to attach to the host debugger. So because we are not calling this function in the KDC for example, there is no way for reading the key database by debugging. This is actually an issue of remote debugging in the embedded system domain and has nothing to do with security features in PikeOS. But for completeness we need to say that for now debugging offers no way to explore keys from the KDC in our PikeOS implementation. Additionally and through the need of remote debugging, a malicious user would also need physical access to the embedded system, which is also not of our concern.

5.7.4.3 Comparison

Besides debugging, PikeOS offers additional features for examining the state and data of the whole system. They will be introduced shortly below. However, if set up correctly they don't introduce a threat to the systems security nor allow reading the data of the KDC.

The *tracing* facility in PikeOS for example allows to trace the occurrance of events in applications, like memory handling, panic events, thread handling, signals, timer events, scheduling, and so on. Tracing can be used to find errors that are difficult to be detected by a conventional debugger. Tracing per default is allowed to tasks in user partitions, see table 5.2. So by enabling the *P4_AB_TRACE* ability in a task's ability mask (see listing 5.5), they are allowed to use kernel services provided by the tracing API. However, the trace API actually offers no functionality for attaching to other user processes nor reading data.

```
struct P4_task_attr_str {
        P4_task_state_t state;
        P4_prio_t mcp;
        P4_uint32_t respart;
        P4_task_t parent;
        P4_ability_mask_t ability_bitmap;
        P4_cpumask_t cpu_mask;
        P4_thr_t max_threads;
        P4_uid_t dnotify;
        char name[P4_NAMELEN];
        P4_interrupt_bitmap_t interrupt_bitmap;
        P4_device_bitmap_t device_bitmap;
};
```

Listing 5.5: PikeOS Task Attributes Struct

Monitoring in terms of PikeOS and the POSIX personality allows to inspect the attributes of process instances. This feature may be compared to reading the */proc* filesystem in Linux. However, the monitoring API also offers no functions for attaching to other tasks or reading data from them. Besides that, tasks in user partitions are not allowed to monitor other tasks, since the *P4_AB_MON* ability per default is not set (see table 5.2). In our setup there are only user partitions, hence actually no task is allowed to monitor other tasks.

The last feature is optional to the POSIX personality and is most similar to *ptrace()* in Linux. The *Instrumentation and Monitoring (Instrumon)* facility uses a PSSW shared memory segment to export an image of the application status which may include information on application threads, thread specific data, timers, message queues, signal actions, and so on. The Instrumon API also offers functions to attach to processes and read or write data to them. But once again, the Instrumon feature has to be enabled by the system developer in both the application and integration project, hence per default is not allowed.

5.7.5 CASE 5: Device Drivers

5.7.5.1 I/O Servers

PikeOS is based on a microkernel architecture. Thereby, as described in chapters 3.5 and 5.7.1, device drivers are not provided by the kernel. Hence they need to be implemented by a higher level application running in user mode. In *PikeOS* a device driver is often implemented as an application that runs in a separate partition. This for example allows a setup where hardware access is concentrated in a dedicated partition running trusted code. Other, possible larger applications then can run in separate partitions with no hardware access. This design also enhances the system's security and robustness. I/O operations of the (untrusted) applications are then delegated to the applications running in the trusted partitions, which in turn perform the actual hardware access. Hence the applications in the trusted partitions act as an *I/O Server* for applications in other partitions.

I/O devices are system resources, the right to access them needs to be configured per process. This is done through the PSSW by reading the setup in the VMIT. When the process that is acting as an I/O server starts, the configured I/O memory resources are mapped into the virtual address space of the process. In this way I/O memory can be directly manipulated by the application's code. Internally I/O servers are implemented as *file providers*, where applications communicate with the I/O server over PikeOS' filesystem. Therefore the application serving I/O operations needs to register as a file provider. This is done by a special entry in the VMIT. Then other applications can access the file system by using the file system API, for example by calling the *vm_open()* function, if they have the corresponding file access rights.

5.7.5.2 File Access

File access privileges are defined for each partition in the VMIT by the *File Access List* attribute, but may be further constrained by a file provider. The different access types are listed below.

- ***VM_O_STAT***: Permit retrieval of file status information.
- ***VM_O_RD***: Permit read access.
- ***VM_O_WR***: Permit write access.
- ***VM_O_RW***: Permit read and write access.
- ***VM_O_EXEC***: Permit execution.
- ***VM_O_RWX***: Permit read access, write access and execution.
- ***VM_O_MAP***: Permit mapping a file into the process's memory address space.
- ***VM_O_FSPROV***: Permit an application to act as an external file provider (mutually exclusive with all others).

5.7.5.3 Comparison

In a monolithic operating system like Linux, most of the services are like memory management, I/O primitives, file systems, device drivers, and network protocol stacks are implemented in the kernel, hence running in privileged mode. In CASE 5 (see chapter 5.4.6) we lined out the major drawbacks of this approach by means of loadable kernel modules and device drivers. When code runs in privileged mode there is no restriction on what it can do. As demonstrated, it is also allowed to access the kernel's task data structures and read their data. It also could potentially violate security policies that are usually implemented in the kernel. Verifying that such code acutually does not violate those policies takes considerable examining and testing effort.

In PikeOS and microkernel architectures in general, most operating system services are pulled out of the kernel and now run in unprivileged mode. Through this the code that was able to violate security policies now is subject to them, which allows a more straightforware and effective system evaluation. This also makes the kernel small so that it can be fully evaluated. To sum up, the implementation of device drivers in PikeOS has following advantages:

- I/O servers run in separate partitions, hence operate on separate sets of resources. So they are completely isolated and there is no way for a program in one partition to programs in other partitions.
- I/O servers now are running in user mode and therefore are subject to kernel security policies.
- In PikeOS there is no feature like LKMs, the kernel cannot be extended.
- For enhanced security, PikeOS allows to further restrict access to the I/O server file system by defining file access rights per process.

CHAPTER 6

Conclusion

In this thesis we worked out that security is a design issue that should be taken into account in the beginning of any project. Retrofitting hardware and software systems to meet security requirments can become quite expensive. By introducing general security terms in the first chapters we found out that *trusted systems* are a fundamental concept for most security architectures. They are founded on the idea to *separate* trusted software, that potentially is verified to be error-free, from untrusted software, that may origin from a malicious user. Software separation can be enforced by implementing *isolation techniques* that partition the system into *protection domains*. Protection domains then act as a logical container for tasks and shared resources. This also allow to split up system-wide security policies into smaller parts which makes policy enforcement simpler. In this context we also mentioned the MILS architecture that was designed to introduce separation and protection domain enforcement into (operating-) systems.

The case study showed us, that Linux in a basic configuration is not able to enforce a strict separation between processes and that some features introduce security issues or may be inappropriate for embedded systems. As opposed to this the PikeOS Partitioning Operating System is able to strictly partition a system by implementing the MILS architecture. Partitions are built through the *separation microkernel* by paravirtulization. Additional security features allow to further restrict the rights of processes, hence may be totally isolated from other software. Since the microkernel is small enough, it has been certified by Common Criteria adding another security assurance level.

Through increasing processing power, partitioning operating systems get more and more interesting for embedded real-time systems. Embedded Hardware nowadays is able to context switch between partitions or virtual machines in reliable time which was not possible some years ago. Through their strict partitioning in time and space they are especially applicable for hard real-time systems.

Bibliography

[ABB$^+$86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[AFS97] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, pages 65–, Washington, DC, USA, 1997. IEEE Computer Society.

[AJ95] Marshall D. Abrams and Michael V. Joyce. Trusted system concepts. *Computers &amp; Security*, 14(1):45 – 56, 1995.

[AKS99] Albert Alexandrov, Paul Kmiec, and Klaus Schauser. Consh: Confined execution environment for internet computations, 1999.

[And72] James P. Anderson. Computer Security technology planning study. Technical report, Deputy for Command and Management System, USA, 1972.

[BB04] Phililip J. Bond and Arden L. Bement. Fips pub 199, 2004.

[BBS95] Andrew Berman, Virgil Bourassa, and Erik Selberg. Tron: Process-specific file protection for the unix operating system. In *In Proceedings of the USENIX 1995 Technical Conference*, pages 165–175, 1995.

[BCP$^+$09] Abhijit Belapurkar, Anirban Chakrabarti, Harigopal Ponnapalli, Niranjan Varadarajan, Srinivas Padmanabhuni, and Srikanth Sundarrajan. *Distributed Systems Security*. Wiley, 2009.

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177, October 2003.

[BGM02] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. Remus: A security-enhanced operating system. *ACM TRANSACTIONS ON INFORMATION AND SYSTEM SECURITY*, 5:2002, 2002.

[Bib77] J. K. Biba. Integrity considerations for secure computer systems, 1977.

[BL73] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical report, MITRE Corporation, March 1973.

[BL76] E. D. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation, 1976.

[Bla93] Matt Blaze. A cryptographic file system for UNIX. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 9–16, New York, NY, USA, 1993. ACM.

[BN89] D. F. C. Brewer and M. J. Nash. The Chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[Bra08] Hans Brandl. Trusted computing grundlagen. In Norbert Pohlmann and Helmut Reimer, editors, *Trusted Computing*, pages 21–42. Vieweg Teubner, 2008.

[BYMX$^+$06] Muli Ben-Yehuda, Jon Mason, Jimi Xenidis, Orran Krieger, Leendert van Doorn, Jun Nakajima, Asit Mallick, and Elsie Wahlig. Utilizing IOMMUs for Virtualization in Linux and Xen. In *OLS '06: The 2006 Ottawa Linux Symposium*, pages 71–86, July 2006.

[Cri12] Common Criteria, 2012.

[CW87] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, 1987.

[DG09] Heradon Douglas and Christian Gehrmann. Secure virtualization and multicore platforms: State-of-the-art report. *Innovation*, page 71, 2009.

[Dis11] Discretix. Cryptocell, 2011. www.discretix.com.

[DMRS97] Asit Dan, Ajay Mohindra, Rajiv Ramaswami, and Dinkar Sitaram. Chakravyuha: A sandbox operating system for the controlled execution of alien code. Technical Report RC20742, IBM T.J. Watson research center, 1997.

[ES00] Ulfar Erlingsson and Fred B. Schneider. Irm enforcement of java stack inspection. In *In IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[FL93] W. Fumy and P. Landrock. Principles of key management. *Selected Areas in Communications, IEEE Journal on*, 11(5):785 –793, jun 1993.

[GPR04] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Symposium*, February 2004.

[GR95] B. Guttman and E.A. Roback. *An Introduction to Computer Security: The Nist Handbook*. NIST special publication. Diane Pub Co, 1995.

[GRS12] GRSecurity. Grsecurity. `http://grsecurity.net`, August 2012.

[GSMB03] Eu-jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. Sirius: Securing remote untrusted storage. In *in Proc. Network and Distributed Systems Security (NDSS) Symposium 2003*, pages 131–145, 2003.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications – confining the wily hacker, 1996.

[IBS02] Sotiris Ioannidis, Steven M. Bellovin, and Jonathan M. Smith. Sub-operating systems: A new approach to application security. In *In Proceedings of the 10th ACM SIGOPS European Workshop*, 2002.

[ISO88] ISO. Open systems interconnection reference model - part2: Security architecture, 1988.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding, 2002.

[KLMR04] Paul Kocher, Ruby Lee, Gary Mcgraw, and Srivaths Ravi. Security as a new dimension in embedded system design. In *Proceedings of the 41st*

Design Automation Conference (DAC '04), pages 753–760. ACM Press. Moderator-Srivaths Ravi, 2004.

[Koo04] P. Koopman. Embedded system security. *Computer*, 37(7):95 – 97, july 2004.

[Kro09] Kirk L. Kroeker. The evolution of virtualization. *Commun. ACM*, 52(3):18–20, 2009.

[Lam69] B. W. Lampson. Dynamic protection structures. In *Proceedings of the November 18-20, 1969, fall joint computer conference*, AFIPS '69 (Fall), pages 27–38, New York, NY, USA, 1969. ACM.

[Lam74] Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8:18–24, January 1974.

[Lam04] Butler W. Lampson. Computer security in the real world. *Computer*, pages 37–46, 2004.

[LVS03] Z. Liang, V.N. Venkatakrishnan, and R. Sekar. Isolated program execution: an application transparent approach for executing untrusted programs. In *Computer Security Applications Conference, 2003. Proceedings. 19th Annual*, pages 182 – 191, dec. 2003.

[Nec97] George C. Necula. Proof-carrying code, 1997.

[NSL$^+$06] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, August 2006.

[OK09] R. Obermaisser and H. Kopetz. *Genesys an Artemis Cross-Domain Reference Architecture for Embedded Systems*. Sudwestdeutscher Verlag Fur Hochschulschriften AG, 2009.

[Ope12] Open Group. POSIX.1-2008 / IEEE Std 1003.1 Standard. `http://pubs.opengroup.org/onlinepubs/9699919799/`, August 2012.

[PO10] Enrico Perla and Massimiliano Oldani. *A Guide to Kernel Exploitation: Attacking the Core*. Syngress Publishing, 2010.

[Pro03] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 18–18, Berkeley, CA, USA, 2003. USENIX Association.

[RRC04] Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. Tamper resistance mechanisms for secure embedded systems. In *Proceedings of the 17th International Conference on VLSI Design*, VLSID '04, pages 605–, Washington, DC, USA, 2004. IEEE Computer Society.

[RRKH04] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. Security in embedded systems: Design challenges. *ACM Trans. Embed. Comput. Syst.*, 3:461–491, August 2004.

[Rus81] John Rushby. Design and verification of secure systems. In *In Eighth ACM Symposium on Operating System Principles*, pages 12–21, 1981.

[Rus84] John Rushby. A trusted computing base for embedded systems. In *Proceedings of the 7th Department of Defense/NBS Computer Security Conference*, pages 294–311, 1984.

[SB08] William Stallings and Lawrie Brown. *Computer security: principles and practice*. Prentice Hall, Upper Saddle River, NJ:, 2008.

[SD10] Tom St. Denis. Libtom projects. `http://www.libtom.org`, October 2010.

[SEL12] SELinux. Security Enhanced Linux. `http://http://selinuxproject.org/page/Main_Page`, August 2012.

[SN05] J.E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32 – 38, may 2005.

[Sta08] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, sixth edition, April 2008.

[SYS12] SYSGO AG. PikeOS. `http://www.sysgo.com`, August 2012.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[UP95] A. User and A. Planning. An Introduction to Computer Security: The NIST Handbook. 1995.

[VN09] Arun Viswanathan and B.C. Neumann. A survey of isolation techniques. 2009.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[Yor03] Richard York. A new foundation for cpu systems security. *ARM Limited*, 2003.

APPENDIX A

Data Structures

A.1 messg_t Struct Reference

The message structure for exchanging data from one process to another using SysV Message Queues.

Data Fields

- int **dest_qid**

  Destination message queue identifier. States where to send the message.
- int **src_qid**

  Source message queue identifier. States where the message came from. Necessary for connectionless communication between processes.
- int **service**

  The service that the KDC needs to process. Will be set by the SKDP, for more details see ***skdp.h*** *(*p. *121).*
- int **key_id**

  Unique key identifier.
- int **data_size**

  Size of sent data in bytes.

- unsigned char **data** [MSG_SIZE_BYTES]

  The actual data. May contain secret cryptograpic keys or Diffie-Hellman public keys.

A.2 mymsgbuf_t Struct Reference

Redefinition of the kernel message queue structure.

Data Fields

- long **mtype**

  *Type of the message. Different types are defined and explained in **skdp.h** (*p. *121).*

- **messg_t message**

  The message packet containing all the data.

A.3 Node_ Struct Reference

Defines the data to be stored in the hashtable.

Data Fields

- struct **Node_** ∗ **next**

  Pointer to next key data record.

- **hashTableIndex key_id**

  Unique key identifier.

- unsigned char **private_key** [KEY_ARRAY_SIZE]

  Secret cryptographic key data.

APPENDIX B

Modules

B.1 setup.h File Reference

Global settings for key size and message queues.

Defines

- #define **MQ_PATH** "."

  Path for KDC's message queue. Is needed for calculating the Queue Identifier.
- #define **MQ_ID_MASTER** 'm'

  Unique identifier for KDC's message queue, needed for calculating the Queue Identifier.
- #define **MQ_PARAMS** (IPC_CREAT | S_IRUSR | S_IWUSR)

  Access privilegies for the message queue.
- #define **KEY_SIZE_BITS** 128

  Defines the secret key size in bits.
- #define **KEY_ARRAY_SIZE** (KEY_SIZE_BITS / 8)

  The array size to store a key in an unsigned char array.
- #define **MSG_SIZE_BYTES** 128

  The maximum size in bytes for the data field in a message.

- #define **HASHTABLESIZE** 21

  The size of the KDC's hashtable for storing the cryptographic keys.

B.2 key_db.h File Reference

The database for storing keys is implemented through a hashtable. The hashtable size is set in **setup.h** (p. 112).

Data Structures

- struct **Node_**

  Defines the data to be stored in the hashtable.

Typedefs

- typedef int **hashTableIndex**

  Defining the type for calculating the hashtable index.

Functions

- **Node** ∗ **insertNode** (**Node** ∗hashTable[ ], **hashTableIndex** key_id, unsigned char ∗private_key)

  Inserts a key data record at the beginning of the list with the calculated hash index from key_id.
- int **deleteNode** (**Node** ∗hashTable[ ], **hashTableIndex** key_id)

  If found, the key data record with key_id is deleted from the hashtable.
- **Node** ∗ **findNode** (**Node** ∗hashTable[ ], **hashTableIndex** key_id)

  Searches for a key data record with key_id in the hashtable and returns it.
- void **printHashTable** (**Node** ∗hashTable[ ])

  Prints the whole key database. For debugging purposes only.

- void **freeHashTable** (**Node** ∗hashTable[ ])

  Deallocates all the memory occupied by the hashtable. Will be called on KDC shutdown.

B.3 message.h File Reference

Data Structures

- struct **messg_t**

  The message structure for exchanging data from one process to another using SysV Message Queues.
- struct **mymsgbuf_t**

  Redefinition of the kernel message queue structure.

Functions

- void **set_type** (**mymsgbuf_t** ∗buf, int type)

  Set the message type in buf.
- void **set_dest** (**mymsgbuf_t** ∗buf, int dest)

  Set the destination qid in buf.
- void **set_src** (**mymsgbuf_t** ∗buf, int src)

  Set the source qid in buf.
- void **set_service** (**mymsgbuf_t** ∗buf, int service)

  Set the message service in buf.
- void **set_key_id** (**mymsgbuf_t** ∗buf, int id)

  Set the key_id in buf.
- void **set_data_size** (**mymsgbuf_t** ∗buf, int size)

  Set the size of data in buf.
- void **set_data** (**mymsgbuf_t** ∗buf, unsigned char ∗data)

  Set the data to be sent in buf.
- int **get_type** (**mymsgbuf_t** ∗buf)

Get the message type in buf.

- int **get_dest** (**mymsgbuf_t** ∗buf)

  Get the destination qid in buf.

- int **get_src** (**mymsgbuf_t** ∗buf)

  Get the src qid in buf.

- int **get_service** (**mymsgbuf_t** ∗buf)

  Get the message service in buf.

- int **get_key_id** (**mymsgbuf_t** ∗buf)

  Get the unique key_id in buf.

- int **get_data_size** (**mymsgbuf_t** ∗buf)

  Get the size of data in buf.

- unsigned char ∗ **get_data** (**mymsgbuf_t** ∗buf)

  Get the data in buf.

- key_t **build_key** (char c)

  This function creates a unique SysV IPC key from a letter passed as a parameter.

- int **create_queue** (key_t key)

  This function creates a SysV message queue identified by an IPC key.

- int **remove_queue** (int qid)

  This function removes a message queue identified by the SysV IPC queue identifier from the kernel address space.

- int **send_message** (int qid, **mymsgbuf_t** ∗qbuf)

  This function sends a message to the queue identified by qid.

- int **receive_message** (int qid, **mymsgbuf_t** ∗qbuf, long type, int msgflg)

  This function reads a message from the queue with the specified identifier.

B.3.1 Detailed Description

The communication system for exchanging data between processes is implemented with SysV Message Queues. The *message* layer is the lowest layer of this communication system. It has all the functions to set and get data items from the message structure. It is also responsible for creating and destroying message queues and to actually send and receive data over the queues. This functionality will be implemented by the corresponding SysV IPC system calls that can be looked in the manpages. The *message* layer will be utilized by the higher level *protocol* and *skdp* layers.

B.3.2 Function Documentation

key_t build_key (char *c*)

This function creates a unique SysV IPC key from a letter passed as a parameter.

Parameters

c	create SysV IPC key from c

Returns

Unique key_t SysV IPC key.

Uses POSIX *ftok()* to generate unique SysV IPC keys that are required for generating message queues. For more details see *ftok()* manpages.

int create_queue (key_t *key*)

This function creates a SysV message queue identified by an IPC key.

Parameters

key	create message queue identifier from SysV IPC key

Returns

Message queue identifier.

Uses POSIX *msgget()* to generate the message queue identifier. For more details *msgget()* manpages.

int remove_queue (int *qid*)

This function removes a message queue identified by the SysV IPC queue identifier from the kernel address space.

Parameters

qid	remove queue with qid

Returns

0 on success, -1 otherwise.

Uses POSIX *msgctl()* to delete message queue from kernel space. For mor details see *msgctl()* manpages.

int send_message (int *qid*, mymsgbuf_t ∗ *qbuf*)

This function sends a message to the queue identified by qid.

Parameters

qid	message queue identifier to send the message to
qbuf	message to send

Returns

0 on success, -1 otherwise.

Uses POSIX *msgsnd()* to send a message to a message queue with qid. For more details see *msgsnd()* manpages.

int receive_message (int *qid*, mymsgbuf_t ∗ *qbuf*, long *type*, int *msgflg*)

This function reads a message from the queue with the specified identifier.

Parameters

qid	read from message queue with qid
qbuf	buffer to store the received message
type	indicates which types of messages to receive
msgflg	defines if recieve waits for message or returns immediately

Returns

On success number of bytes received, if no message received 0, -1 otherwise.

Uses POSIX *msgrcv()* to to read a message from the queue with qid. It filters messages by reading only those with the specified type. For more details *msgrcv()* manpages.

B.4 protocol.h File Reference

Functions

- int **init_queue** (int num)

  Creates a SysV Message Queue.
- int **close_queue** (int qid)

  Removes a SysV Message Queue.
- int **send_data** (int dest, int src, int type, int service, int key_id, int size, unsigned char ∗key_data)

  Sets up a message and passes it down to the message layer for sending.
- int **receive_data** (int dest, **mymsgbuf_t** ∗in_msg, int type, int msgflg)

  Receives messages on the message queue.

B.4.1 Detailed Description

The *protocol* layer lies between the *message* layer and the *skdp* layer. It actually sets up and closes a message queue with a certain identifier. The function ***init_queue()*** *(*p. *119)* may also be used to connect to already existing queues, like e.g. a user program connecting to the KDC. The *protocol* layer is also used to correctly set up the messages and then pass them down to the lower *message* layer for sending. Receiving a message will also be passed down to the *message* layer. The communication protocol is connectionless where the queue identifier of the sender is always sent within a message.

B.4.2 Function Documentation

int init_queue (int *num*)

Creates a SysV Message Queue.

Parameters

num	generate message queue identifier from this integer

Returns

Message queue identifier on success, -1 otherwise.

This function will create a SysV Message Queue out of the number parameter. The resulting queue identifier will always be the same for a specified number, allowing programs to connect to other program's message queues. You may configure the KDC to use the identifier 0 and a user program may use *init_queue(0)* to connect to the KDC. For details see ***create_queue(key_t key)*** *(*p. *116)* in the *message* layer.

int close_queue (int *qid*)

Removes a SysV Message Queue.

Parameters

num	message queue identifier

Returns

0 on success, -1 otherwise.

Removes a queue with the specified queue identifier. For details see ***remove_queue(int qid)*** *(p. 117)* in the *message* layer.

int send_data (int *dest,* int *src,* int *type,* int *service,* int *key_id,* int *size,* unsigned char ∗ *key_data*)

Sets up a message and passes it down to the *message* layer for sending.

Parameters

dest	destination queue identifier
src	source queue identifier
type	the message type
service	the requested service
key_id	unique key identifier
size	data size in bytes
key_data	the actual data

Returns

0 on success, -1 otherwise.

This function sets up the items of a message and passes it down to the *message* layer. The parameters are set by the SKPD layer. Message types and services are defined in **skdp.h** (p. 121). For more details see **send_message()** (p. 117) in the *message* layer.

int receive_data (int *dest,* mymsgbuf_t ∗ *in_msg,* int *type,* int *msgflg*)

Receives messages on the message queue.

Parameters

dest	the message queue to read messages from
in_msg	the message buffer to write the received data to
type	specifies the message type to receive
msgflg	defines wait status of receive

Returns

On success number of bytes received, if no message received 0, -1 otherwise.

This function is passed down to the lower *message* layer. The msgflg specifies if a receive will wait for a message or returns immediately. If **msgflg=0**, receive will block until there is a message with the specified type in the message queue. When read, it returns. If **msgflg=IPC_NOWAIT** the message queue will be scanned for a message with the specified type. If not found, receive will return immediately. For more details see **receive_message**() (p. 117) in the *message* layer.

B.5 skdp.h File Reference

Defines

- #define **MSG_TYPE_DH_KEY** 1

  Message type indicating that the contained data is a Diffie-Hellman key. Not available to user programs, since the DH key exchange will be encapsulated within the SKPD.
- #define **MSG_TYPE_SECRET_KEY** 2

  Message type indicating that the contained data is a secret key generated by the KDC. User programs are allowed to set up their messages only with this type and then specify the message with one of the following services.
- #define **MSG_SERVICE_KEY_REQ_NEW** 3

  Message service for user programs to request a new generated key.
- #define **MSG_SERVICE_KEY_REQ_EXISTING** 4

  Message service for user programs to request a key from the key database at the KDC.
- #define **MSG_SERVICE_KEY_RESPONSE** 5

  Message service for the KDC to accordingly respond with key messages.

Functions

- int **setup_protocol** (int id, const struct ltc_prng_descriptor ∗prng_desc)

Will start the secure key distribution protocol.

- int **tear_down_protocol** (int qid)

  Shuts down the SKDP.
- int **get_master_qid** ()

  Returns the qid of the KDC.
- int **client_secure_send_key_request** (int dest, int src, int service, int key_id, **mymsgbuf_-t** ∗out_msg)

  Sends key requests from user programs to the kdc.
- int **master_receive_requests** (int dest, **mymsgbuf_t** ∗in_msg)

  Function the KDC uses to receive requests.
- int **master_secure_send_key_response** (int dest, int src, int key_id, unsigned char ∗data)

  Function the KDC uses to respond with key-messages.

B.5.1 Detailed Description

This module implements the Secure Key Distribution Protocol (SKDP). The SKDP can be used by user programs to request cryptographic keys from the Key Distribution Center (KDC). User programs may request new or already existing keys by a unique key_-id. The protocol is secure since every key-message will be encrypted using a Diffie-Hellman shared secret that is established prior the sending process. User programs must not care about encryption and decryption, this will be done by the SKPD automatically. The cryptographic stuff will be done by the **libtomcrypt** and **libtomsfastmath** libraries (see [SD10]).

B.5.2 Function Documentation

int setup_protocol (int *id,* const struct ltc_prng_descriptor ∗ *prng_desc*)

Will start the secure key distribution protocol.

Parameters

id	identifier to calculate the message queue id from

prng_desc	libtomcrypt descriptor for pseudo random number generator

Returns

upon success the message queue corresponding with id, -1 ohterwise.

This function will start up the SKDP, that includes the creation of a message queue from the id parameter and registering the PRNG descriptor. The pseudo random number generator is needed by KDC for generating (random) keys. It will also be used by the elliptic curve cryptography stuff that implements the Diffie-Hellman key exchange.

int tear_down_protocol (int *qid*)

Shuts down the SKDP.

Parameters

qid	message queue identifier

Returns

Queue identifier generated from id, -1 otherwise.

Will tear down the SKDP by unregistering the PRNG descriptor and freeing the message queue with qid.

int get_master_qid ( )

Returns the qid of the KDC.

Returns

Upon success, queue identifier of the KDC, -1 otherwise.

Will return the qid of the KDC so that user programs can connect to the KDC. In the current implementation the KDC needs to be configured to have the identifier 0.

int client_secure_send_key_request (int *dest*, int *src*, int *service*, int *key_id*, mymsgbuf_t ∗ *out_msg*)

Sends key requests from user programs to the kdc.

Parameters

dest	destination qid (usually the kdc's qid)
src	source qid
service	message service
key_id	the request key with unique key_id
out_msg	message buffer containing data after the protocol is responding with a key.

Returns

0 upon success, -1 otherwise.

This function may be called if a user program wants to request secret keys from the KDC. Therefore the type in send_data is automatically set to MSG_TYPE_SECRET_KEY, the service is specified by the user whether he wants to request new or existing keys. Possible values are defined in **skdp.h** (p. 121). After this first request the protocol waits for the Diffie-Hellman public key of the KDC and calculates the shared secret upon receipt. Then the protocol waits for the encrypted key-data from the KDC. After the data arrived it will be decrypted with the prior calculated shared secret. Encryption and decryption is done by the symmetric AES cipher, Diffie-Hellman private/public keypairs are generated by Elliptic Curve Cryptography (ECC) mechanisms, both implemented in the *libtomcrypt* library.

int master_receive_requests (int *dest*, mymsgbuf_t ∗ *in_msg*)

Function the KDC uses to receive requests.

Parameters

dest	destination qid
in_msg	the request message from user programs

Returns

On success number of bytes received, if no message received 0, -1 otherwise.

The KDC will use this function to listen for key requests at his message queue with *dest* qid. Receiving is non-blocking. Depending on the type of requests sent within in_msg, the KDC will take the according actions.

int master_secure_send_key_response (int *dest,* int *src,* int *key_id,* unsigned char ∗ *data*)

Function the KDC uses to respond with key-messages.

Parameters

dest	destination qid
src	source qid
key_id	unique key identifier
data	the actual encrypted secret key data

Returns

0 on success, -1 otherwise.

After the KDC received a key request, it will respond accordingly with this function. Therefore it exchanges it's public DH key with the user that is already waiting for it in **client_secure_send_key_request()** (p. 124). After calculating the DH shared secret, the KDC encrypts the key data and sends it to the requesting user program.