MINISTRY FOR DEVELOPMENT OF INFORMATION TECHNOLOGIES AND COMMUNICATIONS OF THE REPUBLIC OF UZBEKISTAN

TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES

Admit to protection
Chief of TE chair
Ph.D. A.M. Eshmuradov
_____
_____2016 y.

BACHELOR'S GRADUATIONAL QUALIFICATION WORK

titled: Analysis of the Mac OS X operating system and the evaluation of protection level

Graduator	_____ Signature	Samadov S.O. (name)
Advisor	_____ signature	Khaitbaev A.F. (name)
SP and LS advisor	_____ signature	Sattorov X.A. (name)
Reviewer	_____ signature	Sharifov R.A. (name)

Tashkent – 2016

MINISTRY FOR DEVELOPMENT OF INFORMATION TECHNOLOGIES AND COMMUNICATIONS OF THE REPUBLIC OF UZBEKISTAN

TASHKENT UNIVERSITY OF INFORMATION TECHNOLOGIES

Faculty Telecommunication technologies chair Telecommunication engineering

Specialization 5311300 - Telecommunication

A P P R O V E D

Chief of chair TE _____
« ___ »_____ 2016 y.

A S S I G N M E N T

for the bachelor's graduational qualification work

Sunnatillo Samadov

(name, surname)

Titled

1. Theme is approved be the university order from 21.12.2015 y. №1368-17

2. Deadline for graduational work 23.05.2016 y.

3. Source data to work Materials extracted from Internet, books, abstract technical descriptions and equipment User Manuals

4. Contents of accounting-explanatory note (list of subjects) 1. Operating systems and their types 2. Mac OS operating system and its basic parameters and characteristics.
3. Evaluating the protection level of Mac OS X
4. Life safety

5. The list of graphics material Presentations demo slides

6. Delivery date 16.01.2016

Advisor _____
signature

Assignment accept _____
signature

7. Advisers on separate sections of graduational work

Section	Adviser	Signature, date	
		Assignment given	Assignment taken
Chapter 1 Chapter 2 Chapter 3	Khaitbaev A.F	15.01.2016 y.	15.01.2016 y.
Chapter 4	Sattorov X.A	23.03.2016 y.	23.03.2016 y.

8. The schedule of work performance

№	Section name	Performance date	Advisor signature
1.	Operating systems and their types	15.02.2016 y.	
2.	Mac OS X operating system and its basic parameters and characteristics	26.03.2016 y.	
3.	Evaluating the protection level of Mac OS X	30.04.2016 y.	
4.	Life safety	21.05.2016 y.	

Graduator _____ «____»_____2016 y.
signature

Advisor_____ «____»_____2016 y.
signature

This graduation thesis is devoted to study operating systems, their architecture, security vulnerabilities and requirements of secure operating systems . The protection status of Mac OS is analyzed, certain suggestions to improve security level of Mac OS are given and its protection level is evaluated.

Human vital activity and ecology issues are also considered.

Ушбу битирув малакавий иши операцион тизимлар, уларнинг тузилиш архитектураси, хавфсизлигининг заиф томонлари ва ҳимояланган операцион тизимларга бўлган талабларни ўрганишга бағишланган. Ушбу ишда Mac OS ҳимояланганлик ҳолати таҳлил қилинган, унинг ҳавфсизлиги кучайтириш учун аниқ таклифлар берилган ва унинг ҳимояланганлик даражаси баҳоланган.

Шунингдек ҳаёт фаолияти хавфсизлиги ва экология масалалари ҳам кўриб чиқилган.

Данная выпускная квалификационная работа посвящена изучению операционных систем, их архитектуры, уязвимости в безопасности и требований к безопасности операционных систем. Проведен анализ статуса защищенности Mac OS, и оценен уровень защищенности.

Также рассмотрены вопросы безопасности жизнедеятельности и экологии.

CONTENTS

INTRODUCTION

Operating systems are the software that provides access to the various hardware resources (e.g., CPU, memory, and devices) that comprise a computer system. Any program that is run on a computer system has instructions executed by that computer's CPU, but these programs may also require the use of other peripheral resources of these complex systems. Consider a program that allows a user to enter her password. The operating system provides access to the disk device on which the program is stored, access to device memory to load the program so that it may be executed, the display device to show the user how to enter her password, and keyboard and mouse devices for the user to enter her password. Of course, there are now a multitude of such devices that can be used seamlessly, for the most part, thanks to the function of operating systems.

Ensuring the secure execution of all processes depends on the correct implementation of resource and scheduling mechanisms. First, any correct resource mechanism must provide boundaries between its objects and ensure that its operations do not interfere with one another. For example, a file system must not allow a process request to access one file to overwrite the disk space allocated to another file. Also, file systems must ensure that one write operation is not impacted by the data being read or written in another operation. Second, scheduling mechanisms must ensure availability of resources to processes to prevent denial of service attacks. For example, the algorithms applied by scheduling mechanisms must ensure that all processes are eventually scheduled for execution. These requirements are fundamental to operating system mechanisms, and are assumed to be provided in the context of this book. The scope of this book covers the misuse of these mechanisms to inadvertently or, especially, maliciously impact the execution of another process.

Security becomes an issue because processes in modern computer systems interact in a variety of ways, and the sharing of data among users is a fundamental use of computer systems. First, the output of one process may be used by other

processes. For example, a programmer uses an editor program to write a computer program's source code, compilers and linkers to transform the program code into a form in which it can be executed, and debuggers to view the executing processes image to find errors in source code. In addition, a major use of computer systems is to share information with other users. With the ubiquity of Internet-scale sharing mechanisms, such as e-mail, the web, and instant messaging, users may share anything with anyone in the world. Unfortunately, lots of people, or at least lots of email addresses, web sites, and network requests, want to share stuff with you that aims to circumvent operating system security mechanisms and cause your computer to share additional, unexpected resources. The ease with which malware can be conveyed and the variety of ways that users and their processes may be tricked into running malware present modern operating system developers with significant challenges in ensuring the security of their system's execution.

The challenge in developing operating systems security is to design security mechanisms that protect process execution and their generated data in an environment with such complex interactions. As we will see, formal security mechanisms that enforce provable security goals have been defined, but these mechanisms do not account or only partially account for the complexity of practical systems. As such, the current state of operating systems security takes two forms: constrained systems that can enforce security goals with a high degree of assurance and general-purpose systems that can enforce limited security goals with a low to medium degree of assurance. First, several systems have been developed over the years that have been carefully crafted to ensure correct (i.e., within some low tolerance for bugs) enforcement of specific security goals. These systems generally support few applications, and these applications often have limited functionality and lower performance requirements. That is, in these systems, security is the top priority, and this focus enables the system developers to write software that approaches the ideal of the formal security mechanisms mentioned above. Second, the computing community at large has focused on function and flexibility, resulting in general-purpose, extensible systems that are

very difficult to secure. Such systems are crafted to simplify development and deployment while achieving high performance, and their applications are built to be feature-rich and easy to use. Such systems present several challenges to security practitioners, such as insecure interfaces, dependence of security on arbitrary software, complex interaction with untrusted parties anywhere in the world, etc. But, these systems have defined how the user community works with computers. As a result, the security community faces a difficult task for ensuring security goals in such an environment.

However, recent advances are improving both the utility of the constrained systems and the security of the general-purpose systems. We are encouraged by this movement, which is motivated by the general need for security in Mac OS X, and this book aims to capture many of the efforts in building security into Mac OS X, both constrained and general-purpose systems, with the aim of enabling broader deployment and use of security function of Mac OS X operating system.

1. OPERATING SYSTEMS AND THEIR TYPES

1.1. Operating systems and their basic parameters and characteristics

The operating system is an essential part of a computer system; it is an intermediary component between the application programs and the hardware. The ultimate purpose of an operating system is twofold: (1) to provide various services to users' programs and (2) to control the functioning of the computer system hardware in an efficient and effective manner.

An operating system (OS) is a large and complex set of system programs that control the various operations of a computer system and provide a collection of services to other (user) programs. The purpose of an operating system involves two key goals:

- Availability of a convenient, easy-to-use, and powerful set of services that are provided to the users and the application programs in the computer system
- Management of the computer resources in the most efficient manner.

Application and system programmers directly or indirectly communicate with the operating system in order to request some of these services. The services provided by an operating system are implemented as a large set of system functions that include scheduling of programs, memory management, device management, file management, network management, and other more advanced services related to protection and security. The operating system is also considered a huge resource manager and performs a variety of services as efficiently as possible to ensure the desired performance of the system.

Figure 1.1 shows an external view of a computer system. It illustrates the programmers and end users communicating with the operating system and other system software.

The most important active resource in the computer system is the CPU. Other important resources are memory and I/O devices. Allocation and deallocation of all resources in the system are handled by various resource

managers of the operating system.

General-purpose operating systems support two general modes of operation: user mode and kernel mode, which is sometimes called the supervisor, protected, or privilege mode. A user process will normally execute in user mode. Some instructions, such as low-level I/O functions and memory access to special areas where the OS maintains its data structures, can execute only in kernel mode. As such, the OS in kernel mode has direct control of the computer system.

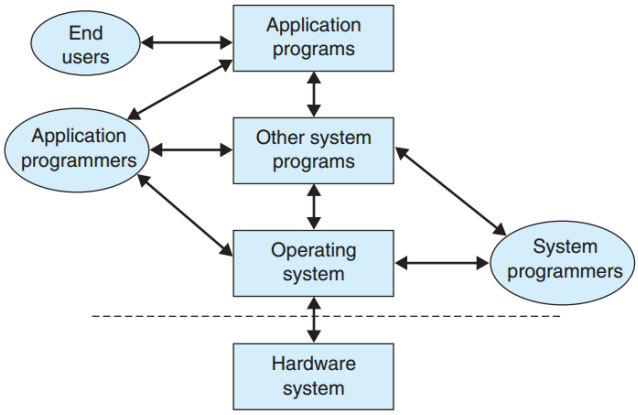

Figure 1.1. An external view of a computer system

Operating System Interfaces

Users and application programmers can communicate with an operating system through its interfaces. There are three general levels of interfaces provided by an operating system:

- Graphical user interface (GUI)
- Command line interpreter (also called the shell)
- System-call interface

Figure 1.2 illustrates the basic structure of an operating system and shows the three levels of interfaces, which are found immediately above the kernel. The highest level is the graphical user interface (GUI), which allows I/O interaction with a user through intuitive icons, menus, and other graphical objects. With these objects, the user interacts with the operating system in a relatively easy and convenient manner—for example, using the click of a mouse to select an option or

command. The most common GUI examples are the Windows desktop and the X-Window in Unix.

The user at this level is completely separated from any intrinsic detail about the underlying system. This level of operating system interface is not considered an essential part of the operating system; it is rather an add-on system software component.

The second level of interface, the command line interpreter, or shell, is a text oriented interface. Advanced users and application programmers will normally directly communicate with the operating system at this level. In general, the GUI and the shell are at the same level in the structure of an operating system. The third level, the system-call interface, is used by the application programs to request the various services provided by the operating system by invoking or calling system functions.

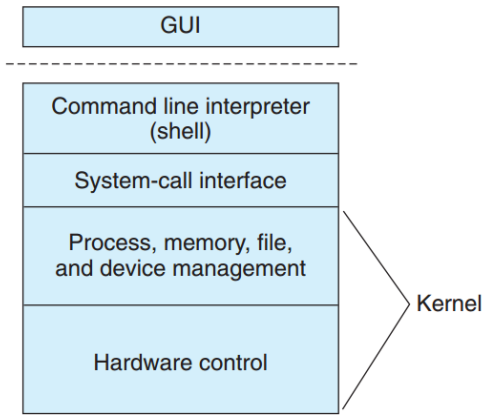

Figure 1.2. Basic structure of an operating system.

Multi-Level Views of an Operating System

The overall structure of an operating system can be more easily studied by dividing it into the various software components and using a top-down (layered) approach. This division includes the separation of the different interface levels, as previously discussed, and the additional components of the operating system. The top layer provides the easiest interface to the human operators and to users interacting with the system. Any layer uses the services or functions provided by

the next lower layer.

As mentioned previously, the main goal of an operating system is to provide services to the various user and system programs. The operating system is structured into several components, each one providing its own group of services. For a more clear understanding of the complexities of how the various services are provided, two abstract views of the OS are presented: (1) the external view and (2) the internal view.

External View of an Operating System

The external view of an operating system is the set of interfaces discussed previously. These are sometimes referred to as the abstract views. Users of a computer system directly or indirectly communicate with the operating system in order to request and receive some type of service provided by the operating system. Users and application programs view the computer system at the higher level(s) of the operating system interface. The low-level OS structures(internal view) and the hardware details are hidden behind the OS interface.

Figure 1.3 illustrates the abstract views of a computer system. Only the upper level abstract views are important to users of the OS. The low-level OS view and the hardware view are relevant only to system specialists.

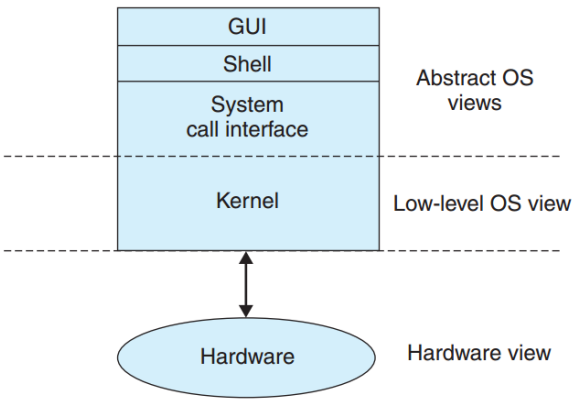

Figure 1.3. Abstract views of a computer system.

One of the main principles applied in the design of the external view concept is simplicity. The system interfaces hide the details of how the underlying (lower-level) machinery operates. The operational model of the computer system

presented to the user should be relatively easy to use and is an abstract model of the operations of the system.

The abstraction provided by the OS interfaces gives users and application programs the appearance of a simpler machine. The various programs also convey the illusion that each one executes on its own machine; this abstraction is some times called the abstract machine view.

Internal View of an Operating System

The system services interface (or the system call interface) separates the kernel from the application layer. Located above the hardware, the kernel is the core and the most critical part of the operating system and needs to be always resident in memory. A detailed knowledge about the different components of the operating system, including these lower-level components, corresponds to an internal view of the system.

In studying the internal view of the operating system, it becomes important to identify the various components of the OS that establish policies for the various services they support and the mechanisms used to implement these policies.

It was previously noted that the various services provided by the operating systems include process management, memory management, device management, file management, network management, and other more advanced services related to protection and security. The most important functional components of the operating system are as follows:

- Process manager
- Memory manager
- Resource manager
- File manager
- Device manager

These components are actually modules of the operating system and are strongly related. In addition to these basic functions, most operating systems have some type of network management, security management, and other more advanced functions.

Process Manager

The *process manager* can be considered the most important component of the operating system. It carries out several services such as creating, suspending (blocking), executing, terminating, and destroying processes.

With multiprogramming, several active processes are stored and kept in memory at the same time. If there is only one CPU in the system, then only one process can be in execution at a given time; the other processes are waiting for CPU service. The decision regarding which process to execute next is taken by the scheduler. The dispatcher allocates the CPU to a process and deallocates the CPU from another process. This switching of the CPU from one process to another process is called *context switching* and is considered overhead time.

Memory Manager

The *memory manager* controls the allocation and deallocation of memory. It imposes certain policies and mechanisms for memory management. This component also includes policies and mechanisms for memory protection. Relevant memory management schemes are paging, segmentation, and virtual memory.

Resource Manager

The *resource manager* facilitates the allocation and deallocation of resources to the requesting processes. Functions of this component include the maintenance of resource tables with information such as the number of resources of each type that are available and the number of resources of each type that have been allocated to specific processes.

File Manager

The *file manager* allows users and processes to create and delete files and directories. In most modern operating systems, files are associated with mass storage devices such as magnetic tapes and disks. Data can be read and/or written to a file using functions such as open file, read data from file, write data to file, and close file. The Unix operating system also uses files to represent I/O devices (including the main input device and a main output device such as the console

keyboard and video screen).

Device Manager

For the various devices in the computer system, the device manager provides an appropriate level of abstraction to handle system devices. For every device type, a device driver program is included in the OS. The device manager operates in combination with the resource manager and file manager. Usually, user processes are not provided direct access to system resources. Instead, processes request services to the file manager and/or the resource manager.

Categories of Operating Systems

A common way to classify operating systems is based on how user programs are processed. These systems can be classified into one of the following categories:

- *Batch systems*, in which a set of jobs is submitted in sequence for processing. These systems have a relatively long average turnaround period (interval from job arrival time to completion time) for jobs.
- *Interactive systems*, which support interactive computing for users connected to the computer system via communication lines. The most common type of operating systems that support interactive computing is *time-sharing*. Figure 1.4 illustrates the general structure of a time-sharing system. A time-sharing system is an example of a multiuser system, which means that multiple users are accessing the system at the same time.
- *Real-time systems*, which support application programs with very tight timing constraints.
- *Hybrid systems*, which support batch and interactive computing. The most general classification of operating systems is regarding the application environment and includesthe following categories:
- *General purpose*, which are used mainly to develop programs and execute these on most types of platforms.

- *Application-dependent*, which are designed for a special purpose. An example of this type is a real-time operating system that controls a power plant for the generation of electricity.

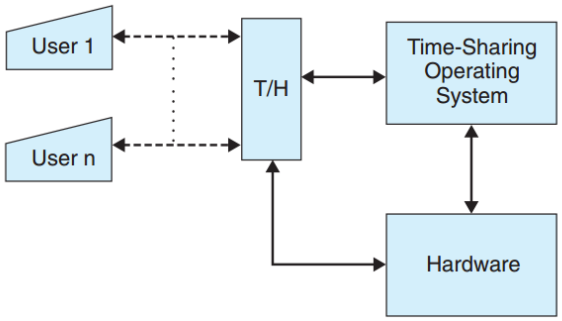

Figure 1.4. General structure of a time-sharing system

Systems can also be classified as being single user or multiuser. In single-user systems, the single user has access to all resources of the system all the time. In multiuser systems, several users request access to the resources of the system simultaneously.

The term *job* refers to a unit of work submitted by a user to the operating system. A typical job consists of the following parts:

- A sequence of commands to the operating system
- A program either in a source language or in binary form
- A set of input data used by the program when it executes

The sequence of commands in a job includes commands for compiling the source program provided (with an appropriate compiler), linking the program with appropriate libraries, loading the linked program, and executing the program with the input data provided. The term *job* was used in the early batch operating systems. The term *process* basically refers to an execution instance of a program.

Modern Operating Systems

The three most widely used operating systems are Linux, Unix, and Windows. These actually represent two different families of operating systems that have evolved over the years: Unix/Linux and Microsoft Windows.

Unix

Unix was originally introduced in 1974 by Dennis Ritchie and Ken Thompson while working at AT&T Bell Labs. The operating system was developed on a small computer and had two design goals: small size of the software system and portability. By 1980, many of the users were universities and research labs. Two versions became the best known systems: System V Unix (AT&T) and BSD Unix (University of California at Berkeley).

Unix became the dominant time-sharing OS used in small and large computers. It is one of the first operating systems written almost entirely in C, a high-level programming language.

The Unix family includes Linux, which was primarily designed and first implemented by Linus Torvals and other collaborators in 1991. Torvals released the source code on the Internet and invited designers and programmers to contribute their modifications and enhancements. Today Linux is freely available and has been implemented in many small and large computers. This operating system has also become important in commercial software systems.

Microsoft Windows

Windows operating systems are considered the most widely used family of operating systems released by Microsoft. These systems were developed for personal computers that use the Intel microprocessors. This group of operating systems consists of Windows 95, 98, ME, and CE, which can be considered the smaller members of the family. The larger and more powerful members of the Windows family of operating systems are WindowsNT, 2000, XP, Vista, and Windows7. The larger Windows operating systems include a comprehensive security model, more functionality facility for network support, improved and more powerful virtual memory management, and full implementation of the Win32 API functions.

Mac OS X

Mac OS X is an operating system that is really a very good development platform. Mac OS X is based on BSD (Berkeley System Distribution) UNIX and

provides support for many POSIX, Linux, and System V APIs. Apple integrated the widely used FreeBSD 5 UNIX distribution with the Mach 3.0 microkernel. This OS supports multiple development technologies including UNIX, Java, the proprietary Cocoa and Carbon runtime environments, and several open source, web, scripting, database, and development technologies. Mac OS X provides several runtime environments integrated under a single desktop environment. The system provides an object-oriented application framework, procedural APIs, a highly-optimized and tightly integrated implementation of Java Standard Edition (JSE), BSD UNIX API sand libraries, and X11.

1.2. Requirements to ensure the security of the operating systems

Most modern information computer systems provide concurrent execution of multiple applications in a single physical computing hardware (which may contain multiple processing units). Within such a multitasking, time-sharing environment, individual application jobs share the same resources of the system, e.g., CPU, memory, disk, and I/O devices, under the control of the operating system. In order to protect the execution of individual application jobs from possible interference and attack of other jobs, most contemporary operating systems implement some abstract property of containment, such as process (or task) and TCB (Task Control Block), virtual memory space, file, port, and IPC (Inter Process Communication), etc. An application is controlled that only given resources (e.g., file, process, I/O, IPC) it can access, and given operations (e.g., execution or read-only) it can perform.

However, the limited containment supported by most commercial operating systems (MS Windows, various flavors of Unix, Mac OS etc) bases access decisions only on user identity and ownership without considering additional security-relevant criteria such as the operation and trustworthiness of programs, the role of the user, and the sensitivity or integrity of the data. As long as users or applications have complete discretion over objects, it will not be possible to control data flows or enforce a system-wide security policy. Because

of such weakness of current operating systems, it is rather easy to breach the security of an entire system once an application has been compromised, e.g., by a buffer overflow attack. Some examples of potential exploits from a compromised application are :

- Use of unprotected system resources illegitimately. For example, a worm program launches attack via emails to all targets in the address book of a user after it gets control in a user account.
- Subversion of application enforced protection through the control of underneath system. For example, to deface a Web site by gaining the control of the Web server of the site, say changing a virtual directory in Microsoft IIS.
- Gain direct access to protected system resources by misusing privileges. For example, a compromised "sendmail" program running as root on a standard Unix OS will result in super user privileges for the attacker and uncontrolled accesses to all system resources.
- Furnish of bogus security decision-making information. For example, spoof of a file handle of Sun's NFS may easily give remote attackers gaining access to files on the remote file server.

It is not possible to protect against malicious code of an application using existing mechanisms of most commercial operating systems because a program running under the name of a user receives all of the privileges associated with that user. Moreover, the access controls supported by the operating systems are so coarse - only two categories of users: either completely trusted super users (root) or completely un-trusted ordinary users. As the result, most system services and privileged applications in such systems have to run under root privileges that far exceed what they really needed. A compromise in any of these programs would be exploited to obtain complete system control.

Generally, in an access control based security model, there are set of objects, and set of subjects (a subject itself can also be an object). Every object has an associated security attribute, or security label; every subject also has a

security label, or security clearance; and a defined set of control rule, or security policy that dictates which subject is authorized to access which object.

In its effort to address computer security safeguards that would protect classified information in remote-access, resource-sharing computer systems, the National Computer Security Center (NCSC), later DOD (Department of Defense),published an official standard called “Trusted Computer System Evaluation Criteria”, universally known as “the Orange Book”.

The Orange Book defines fundamental security requirements for computer systems and specifies a series of criteria for various levels of security ratings of a computer system based on its system design and security feature. A brief summary of all the ratings and their main characteristics are given as follows with a basic condition that each subsequent higher ratings must meet all the requirements of its lower ones.

D - Minimal Protection: no security is required; the system did not qualify for any of the higher ratings.

C1 - Discretionary Security Protection: the system must identify different users (or jobs) running inside the system, and provide mechanisms for user authentication and authorization to prevent unprivileged user programs from interfere each other (e.g., overwriting critical portions of the memory).

C2 - Controlled Access Protection: the system meets additional security requirements than that of C1 that include access control at a per user granularity (access control for any subset of the user community); clearing of newly allocated disk space and memory; and ability of auditing (logging) for security-relevant events such as authentication and object access, etc.

B1 - Labeled Security Protection: the system must implement the Mandatory Access Control in which every subject and object of the system must maintain a security label, and every access to system resource (objects) by a subject must check for security labels and follow some defined rules.

B2 - Structured Protection: few new security features are added beyond B1; rather the focus is on the structure (design) of the system to maintain greater

levels of assurance so that the system behaves predictably and correctly (such as, a minimal security kernel, trusted path to user, and identified covert channels, etc).

B3 - Security Domains: more requirements to maintain greater assurance that the system will be small enough to be subjected to analysis and tests, and not to have bugs that might allow something to circumvent mandatory access controls, e.g., support of active audit, and secure crashing, etc.

A1 - Verified Design: no additional features in an A1 system over a B3 system; rather there are formal procedures for the analysis of the design of the system and more rigorous controls on its implementation.

Most existing commercial operating systems are with the ratings of C2 or below.

As discussed above, most current operating systems provide discretionary access control, that is, someone who owns a resource can make a decision as to who is allowed to use (access) the resource. Moreover, because the lack of built-in mechanisms for the enforcement of security policies in such systems, the access control is normally a one-shot approach: either all or none privileges are granted, rarely supporting the "principle of least privilege" (without limiting the privileges a program can inherit based on the trustworthiness).

The basic philosophy of discretionary controls assumes that the users and the programs they run are the good guys, and it is up to the operating system to trust them and protect each user from outsiders and other users. Such perception could be extremely difficult to hold true and no longer be considered as secure enough for computer systems of "information era" with broad connectivity through the Internet and heavily commercialization of e-commerce services. Systems with stronger security and protection will require evolving from the approach of discretionary control towards the concept of mandatory (non-discretionary) control where information is confined within a "security perimeter" with strict rules enforced by the system about who is allowed access to certain resources, and not allow any information to move from a more secure

environment to a less secure environment.

Some of basic criteria or requirements of a secure operating system are discussed below.

Mandatory security - a built-in mechanism or logic within the operating system (often called system security module or system security administrator) that implements and tightly controls the definition and assignment of security attributes and their actions (security policies) for every operation or function provided by the system. Generally, a mandatory security will require:

- A policy independent security labeling and decision making logics. The operating system implements the mechanism, whereas the users or applications are able to define security policies.
- Enforcement of access control for all operations. All system operations must have permission checks based on security labeling of the source and target objects. Such enforcement requires controlling the propagation of access rights, enforcing fine-grained access rights and supporting the revocation of previously granted access rights, etc.
- The main security controls include permission or access authorization, authentication usage, cryptographic usage, and subsystem specific usage, etc.

Trusted path - amechanism bywhich atrustworthiness relationshipis established among users and application software so that:

- A user or application may directly interact with trusted software, which can only be activated by either user or trusted software
- Mutually authenticated channel is needed to prevent impersonation of either party.
- The mechanism must be extensible to support subsequent addition of trusted applications.

Support of diverse security policies - traditional MAC mechanisms (such as the multi-level security - MLS [8]) are usually based its security decisions strictly on security clearances for subjects and security labels for objects (see Section 3),

and are normally too restricted to serve as a general security solution. A secure architecture requires flexibility for support of a wide variety of security policies:

- Separation of security policy logic from the mechanism of policy enforcement, so that a system can support diverse security policies.
- Support for policy definition and policy changes with well-defined policy interfaces and formats.
- Provide of default security behavior of the system so that to maintain tight system security without requiring detailed system configuration.

Assurance - a process or methodology to verify the design and implementation of the system that should actually behave as it claims to be and meet the security requirements:

- The process generally involves two elements, (i) statement of the security properties a system is claimed to satisfy; and (ii) some kind of argument or evidence that the system does satisfy those properties.
- The structure of such systems normally requires a small security kernel or module so that the system behavior would relatively easy be verified.
- One of the concerns for a secure operating system is the so-called covert channels, which are the means to circumvent the security barrier enforced by the system in prevention of passing information from one security domain to a less secure domain. For example, one possible covert channel is a “timing channel”, where a Trojan horse program alternately loops and waits, in cycles of, say one minute per bit, and a program outside the perimeter that constantly tests the loading of the system may sense the information the Trojan horse intended to send. There is no general way to prevent all covert channels. It is more practical to introduce enough noise or reduce the bandwidth of such channels in the system so that they won’t be useful to an intruder.

The efforts for the development of secure operating systems can be dated to the earlier days of operating system development (e.g., Multics and Hydra).

With the rapid growth of Internet connectivity and e-commerce, recent

development of secure operating systems spreads from traditional focus of defense or military related systems to more general commercial systems. As a case study, next section presents detailed discussions of a publicly available secure system from National Security Agency (NSA).

1.3. Observations on Common Security Vulnerabilities

Observations on common operating system vulnerabilities, which are primarily derived from data collected in our security analysis summarized and are presented in this section. Our ambition is *not to* suggest yet another vulnerability taxonomy but rather to emphasize common weaknesses that have been exploited in real intrusions. Similar vulnerabilities have been grouped together in a heuristic manner rather than according to some formal model. A collection of five common security problems has been identified and will be further described below. These are:

- Improper input validation
- Weak cryptographic algorithms
- Weak authentication protocols
- Insecure bootstrapping
- Configuration mistakes

The first four of these have a technical or system-related basis, while the latter is related to organizational problems or management. This implies that they are not orthogonal, i.e., a specific vulnerability may belong to more than one group. Apart from describing the security problems, we also put them in relation to the taxonomies. A few concrete examples illustrating the vulnerabilities are also given.

Not all vulnerabilities reported in the appended papers originate from the operating system itself. A badly chosen password, for example, is not an operating system vulnerability, but rather a user related weakness, caused by the fact that the user is either unaware of the possible consequences or just careless.

Improper input validation. In general, it is essential to carefully check the

input to software routines, i.e., to perform an input validation. The check may be with regard to the number of parameters provided, the type of each parameter, or to simply ensure that the amount of input data is not larger than the buffer allocated to store the data. Improper or non-existent input validation is a well-known and serious problem in operating systems. Both Landwehr *et al*. and Aslam address these kinds of flaws in their taxonomies.

Example 1. Probably the best known example of an input validation vulnerability is the buffer overflow exploited in the Internet Worm. In this case, the *fingerd* program was attacked. However, the vulnerability exploited was located in the gets routine in the standard C language I/O library1, which did not perform any validation at all of input bounds.

Example 2. Parameter validation in system functions is especially important, since these execute in kernel mode. However, not all operating systems perform proper control of system call parameters, e.g., Windows NT. This is demonstrated in the *NTCrash* utility.

Example 3. Buffer overflows in protocol implementations have also been very common in many of today's network based operating systems. During the last couple of years a large number of exploit scripts demonstrating packet handling flaws have been published on the Internet. *Teardrop, bonk, boink, Land,* and *LaTierra* are all examples of such scripts. Apparently, many router products on the market were also sensitive to these attacks.

Weak cryptographic algorithms. Overly weak algorithms in cryptographic systems are, in our opinion, another security problem. In operating systems, cryptographic algorithms have long been used to encrypt passwords. However, if the algorithm in use is not sufficiently strong, an attacker may manage to derive plaintext passwords from its corresponding encrypted representation. Note that a strong cryptographic algorithm can suddenly become weak as a result of a research break through in, for example, number theory.

Example 4. In Windows NT, two hashed variants of each password are managed, one in Lan Manager (LM) and one in NT native format. The Lan

Manager password is padded with NULLs if it is less than 14 characters long. This together with the encryption method used creates recognizable patterns if the password contains less than eight characters, which dramatically reduces the number of trials necessary in a brute force attack.

Example 5. Encrypted passwords in standard UNIX were stored in the public readable file, /etc/passwd. In the first password algorithm, the password was used as the key, and a constant was encrypted using this key. The result from the algorithm is the encrypted password. However, if more than one user have selected the same password, they will also have exactly the same encrypted password stored in /etc/passwd, which might lead to accidental discovery of the other users's password. This was not satisfactory. The solution to this problem , was to introduce something that was called "salt". The salt is a 12-bit random number, obtained from the real-time clock when a password is first entered. This random number is appended to the typed password before it is encrypted. The result from this encryption together with the salt is then stored in /etc/passwd. Hence, there are 4096 (2^{12}) encrypted versions of each password. This does not increase the amount of time required to search for a single user's password, but it becomes more or less unfeasible to prepare an encrypted dictionary in advance, thus preventing parallel searches of many passwords simultaneously. An in-depth presentation of the UNIX encrypted password system is found in.

Landwehr *et al.* address this type of vulnerability as a subcategory called *identification/authentication* inadequate in the genesis dimension, and a subcategory called *identification/authentication* in the location dimension. Aslam, on the other hand, does not cover it, although the extended and revised version suggested by Krsul has a subcategory called *authentication.* Below, we illustrate weak authentication protocols with three *manin-the-middle* attacks. They are thoroughly described in.

Examples 6 and 7. Two of the attacks are indeed very similar. The first is directed towards a remote logon in Windows NT, while the other is on the Network Information System (NIS). The latter is often used in a networked UNIX

environment to facilitate centralized administration of the system. In both cases, the attacker manages to fool a victim client by masquerading as the server. Both attacks succeeded because of misplaced trust, i.e., the clients blindly trust the information without verifying that the server is the actual sender.

Example 8. Windows NT supports eight different authentication variants with different security levels. The reason for this is backward compatibility. The client and the server negotiate over which authentication method should be used. Since the server will by default accept plaintext passwords as a valid authentication variant, it is sometimes possible for an attacker to fool a client into using a weaker authentication variant than necessary by interfering with the dialogue between the negotiating parties and to intercept the password later while it is being transmitted. None of the parties will know that such a downgrading attack was performed.

Insecure bootstrapping. From our security analyses it is apparent that system initialization (i.e., bootstrapping) is a major security problem in today's operating systems. All studied systems were vulnerable during bootstrapping. Landwehr *et al.* use a subcategory called *system initialization* in the location dimension, while Aslam regards system initialization problems as an *environment fault.*

Example 9. In the first full-scale experiment [Paper B], many attackers discovered that the SunOS was easily rebooted in single-user mode. Commands entered in that mode run with root privileges , and it was possible to extend these privileges to the server, i.e., the whole network.

Example 10. A Windows NT system executing on a PC can most often be rebooted with a foreign operating system, such as MS-DOS. Once a foreign system is booted on the PC, the NTFS volume may be mounted. Access to files on the newly mounted volume will bypass the access control mechanism enforced when Windows NT is operating.

Configuration mistakes. In current operating systems, security features and mechanisms are seldom activated by default. A secure "out-of-the-box" installation is the exception rather than the rule. To achieve an acceptable security

level, the system owner must, after he has completed installation, spend quite some time in securing the system. However, securing a system is not a trivial task, since operating systems are large and complex. In addition, the number of skilled practitioners in computer security is very small . In this work, vulnerabilities introduced during installation and/or configuration are collectively referred to as configuration mistakes. However, Aslam divided configuration flaws into three subcategories, *object installed with incorrect permissions, utility installed in the wrong place, and utility installed with incorrect setup parameters.* In the taxonomy proposed by Landwehr *et al.,* configuration mistakes are covered in the time of introduction dimension as a subcategory, i.e., *during maintenance.*

In the first SunOS experiment the NIS server configuration database directory /var/yp had permission mode 0777, i.e.,

```
drwxrwxrwx 5 root staff Sep 30 2015 512 yp
```

This implies that the owner, the group, and all others were allowed to read, write, and execute the directory. Such a configuration is very dangerous, since anyone can add a new user or group, or delete a user or group.

Example 12. In earlier NetWare systems , it was not difficult to forge NCP packets transmitted between clients and server(s). Therefore, Novell introduced the concepts of packet signatures in version 3.12. Unfortunately, this feature was not activated by default. In a default installation of the most recent version of NetWare, i.e., version 5, packet signatures will be used if requested by the other side. This implies that, if both the clients and the server(s) use a default installation, packets will not be signed.

Summary

The two basic components of a computer system are the hardware and the software components. An operating system is a large and complex software component of a computer system that controls the major operations of user programs and manages all the hardware resources. The main type of system studied in this book is the general purpose operating system. This chapter has

presented the fundamental concepts of operating systems. Discussions on the general structure, user interfaces, and functions of an operating system have also been presented. Two examples of well-known operating systems are Mac OS and Microsoft Windows.

The abstract machine view is seen by the programs that request some type of service from the operating system. The internal details of the OS are hidden below the interfaces. The internal view includes the components that represent the modules of the OS for carrying out various functions: the process manager, memory manager, resource manager, file manager, and device manager. The process manager is considered the most important component of the OS.

2. MAC OS X OPERATING SYSTEM AND ITS BASIC PARAMETERS AND CHARACTERISTICS

2.1. Mac OS X System Technologies

Mac OS X is both a radical departure from previous Macintosh operating systems and a natural evolution from them. It carries on the Macintosh tradition of ease-of-use, but more than ever it is designed not only to be easy to use but a pleasure to use.

This next-generation operating system is a synthesis of technologies; some are new and most are standard in the computer industry. It is firmly fixed on the solid foundation of a modern core operating system, bringing benefits such as protected memory and preemptive multitasking to Macintosh computing. Mac OS X has a sparkling user interface capable of visual effects such as translucence and drop shadows. These effects, as well as the sharpest graphics ever seen on a personal computer, are made possible by a graphics technology that Apple developed specifically for Mac OS X.

But Mac OS X is more than a sophisticated core and a pretty face. With its multiple application environments, virtually all Macintosh applications can run on it. And with its support for many networking protocols and services, Mac OS X is the ultimate platform for using and enjoying the Internet. It also offers a high degree of interoperability with other operating systems because of its multiple volume formats and its conformance with established and evolving standards.

From a functional perspective, these are the most important components of Mac OS X:

- Aqua, the human-interface design behind the user's experience the application environments Carbon, Cocoa, Java, and Classic
- the windowing and graphics system, as implemented by Quartz (and which includes support for QuickTime and OpenGL)
- Darwin, the advanced, UNIX-based core of the operating system

Figure 2.1 depicts the general dependencies between these components. The rest of this chapter describes what these and other technologies of Mac OS X have to offer.

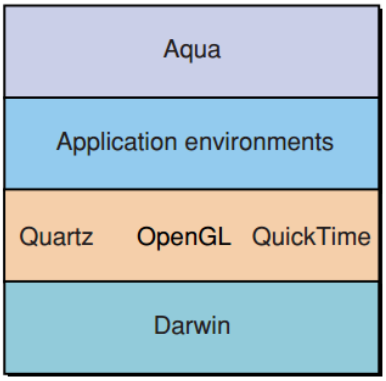

Figure 2.1. A functional view of Mac OS X

The User Experience

The user environment for Mac OS X is similar to that of earlier versions of the Mac OS. But it is also different in important ways. There are differences in the design of the user interface, in the infrastructure for localizing the interface, and in the way to add application features. Mechanisms for exporting and accessing the services of other applications have been enhanced.

And, of course, the user experience draws from the benefits obtained through the core of the operating system. A Macintosh computer remains stable even when an application crashes, and no single application or task can monopolize processing resources; applications can execute concurrently.

This section describes the experience that Mac OS X offers to users and the features and applications that make the experience a productive and enjoyable one.

Aqua

When Apple designed Aqua, the graphical user interface for Mac OS X, it had one goal in mind: to create a modern operating system that is not only easy to use, but is more appealing than any Mac OS you've ever seen As "aqua" suggests, the properties of water infuse the lucid appearance of Mac OS X. Aqua brings a computer to life with color, depth, clarity, translucence, and motion. Buttons look

like polished blue gems, active buttons pulse, windows have drop shadows to give them depth, minimized windows swoop into their Dock icon like a genie into its bottle.

The Finder

A big part of the Aqua experience for users is the design of the desktop and the Finder, a system application that acts as the primary interface for file-system interaction. Users are likely to notice two major innovations in this area: the Dock and the way the Finder displays the elements of the file system.

The Dock reduces desktop clutter. This area of the screen holds just about anything you want to keep handy for instant access: folders, applications, documents, storage devices, minimized windows, QuickTime movies, links to websites. An icon identifies each item stored in the Dock; these icons often provide useful feedback about what they represent. For example, the icon for Mail tells you if you have any new messages waiting to be read. If you store an image, the Dock shows it in preview mode, so you can tell what it is without opening it. And because you can minimize running applications into the Dock, a quick look at the bottom of the screen tells you what applications you're currently running. To switch between tasks, simply click the application or document icon you want to start using, and it becomes the new active task. If you don't know what an icon represents, you can move your mouse pointer over it and the title of the document, folder, or application appears.

Application Support

Part of the Mac OS X user experience is the seamless interaction among different components of the operating system. From BSD to QuickTime, Mac OS X consists of technologies with widely different histories and based on different standards and conventions. A single Mac OS X system hosts volumes of different formats, supports different network file-sharing protocols, and can run applications based on radically different APIs.

Mac OS X provides an easy transition for users and developers. To this end, Mac OS X supports four application environments, each intended for a particular type of application:

- The Classic environment can run most Mac OS 9 applications. Because Classic is a compatibility environment, it does not support some Mac OS X features, such as Aqua or core architectural enhancements provided by Darwin.
- The Carbon environment runs all Mac OS 9 applications whose code has been optimized for Mac OS X. By converting their code to use the Carbon APIs, application developers can ensure that applications take advantage of protected memory, preemptive multitasking, and other features of Darwin.
- The Cocoa environment offers an advanced object-oriented framework for creating the best next-generation applications.
- The Java environment runs 100% Pure Java and mixed-API Java applications and applets.

Mac OS X makes it possible to copy (or cut) almost any piece of data and paste it into an application executing in another environment. It also enables dragging of Finder objects (and the data they represent) between most environments. Mac OS X performs all necessary conversions when, for example, a file stored on a Mac OS Extended (HFS+) volume is copied to a UFS volume.

Multiple Users

Users work on a Mac OS X system in a personally customized environment. They can select a desktop pattern, their preferred language, the applications to start up at boot-time, and a number of other preferences. Whenever they log in to their account, all of their choices are restored.

A user's personalized environment is potentially one of many such environments. Other users can log in to the same computer and have an entirely different set of preferences define their computing environment. Mac OS X enforces secure boundaries between one user's data and programs and another's.

Each account is password-protected and users cannot execute applications or edit or even read documents in another user's folder without the owner's permission. The system gives each user's folder (and all it contains) a default set of permissions that the user can thereafter change to restrict access or grant greater access to other users.

Internationalization

Mac OS X makes it easy to internationalize software. And it does so in such a way that a single binary can support localizations for multiple languages and regional dialects. It also lets software developers dynamically add localized resources for new languages or regions.

Mac OS X includes comprehensive technology to handle text systems used around the world. This text system provides Unicode, input methods, and general text handling services. In Mac OS X, most software comes in the form of a bundle, of which an application is just one type. A bundle is an opaque directory in the file system that contains one or more executables and the resources that go with those executables. One of the primary benefits of bundles is the infrastructure they provide for localizing software. For users, a bundle appears to be a single file object that can be double-clicked or dragged from folder to folder.

Accessibility

Millions of people have some type of disability or special need. Federal regulations in the United States stipulate that computers used in government or educational settings must provide reasonable access for people with disabilities. Mac OS X includes built-in functionality designed to accommodate users with special needs. It also provides software developers with the functions they need to support accessibility in their own applications.

Mac OS X offers the following accessibility support:

- Zoom features let users increase the size of onscreen elements.
- Sticky keys let users press keys sequentially instead of simultaneously, so that they can use keyboard shortcuts.
- Mouse keys let users control the mouse with the numeric keypad.

- Full keyboard-access mode lets users complete any action using the keyboard instead of the mouse.
- Speech recognition lets users speak commands rather than type them.
- Text-to-speech reads text to users with visual disabilities.

In addition to its built-in support, software developers can use Carbon and Cocoa APIs to communicate accessibility information to other applications. Cocoa controls implement the NSAccessibility protocol, which communicates accessibility information to the system. In Carbon there are functions that provide similar support.

Darwin

Beneath the appealing, easy-to-use interface of Mac OS X is a rock-solid, UNIX-based foundation that is engineered for stability, reliability, and performance. This foundation is a core operating system commonly known as Darwin. Darwin integrates a number of technologies, most importantly Mach 3.0, operating-system services based on 4.4BSD (Berkeley Software Distribution), high-performance networking facilities, and support for multiple integrated file systems. Because the design of Darwin is highly modular, you can dynamically add such things as device drivers, networking extensions, and new file systems.

Mach

Mach is at the heart of Darwin because it performs a number of the most critical functions of an operating system. Much of what Mach provides is transparent to applications. It manages processor resources such as CPU usage and memory, handles scheduling, enforces memory protection, and implements a messaging-centered infrastructure for untyped interprocess communication, both local and remote. Mach brings many important advantages to Macintosh computing:

- **Protected memory**. The stability of an operating system should not depend on all executing applications being good “citizens” by not writing data to each others’s (or the system’s) address space; doing so can result in loss or corruption of information and can even precipitate

system crashes. Mach ensures that an application cannot write on another application's memory or on the operating system's memory. By walling off applications from each other and from system processes, Mach makes it virtually impossible for a single poorly behaved application to hurt the rest of the system. And, perhaps best of all, if an application crashes, it doesn't affect the rest of the system and so you don't need to restart your computer.

- **Preemptive multitasking**. In a modern operating system, processes share the CPU efficiently. Mach watches over the computer's processor, prioritizing tasks, making sure activity levels are at the maximum, and ensuring that every task gets the resources it needs. It uses certain criteria to decide how important a task is, and therefore how much time to allocate to it before giving another task its turn. Your process is not dependent on another process yielding its processing time.
- **Advanced virtual memory**. Like other virtual memory systems, Mach maintains address maps that control the translation of a task's virtual addresses into physical memory. Typically only a portion of the data or code contained in a task's virtual address space is resident in physical memory at any given time. As pages are needed, they are loaded into physical memory from storage. Mach augments these semantics with the abstraction of memory objects. Named memory objects enable one task (at a sufficiently low level) to map a range of memory, unmap it, and send it to another task. This capability is essential for implementing separate execution environments on the same system. In Mac OS X, virtual memory is "on" all the time.
- **Real-time support**. This feature guarantees low-latency access to processor resources for time-sensitive media applications.

Darwin also enables cooperative multitasking and preemptive and cooperative threading.

BSD

Integrated with Mach is a customized version of the BSD operating system (currently 4.4BSD). Darwin's implementation of BSD includes much of the POSIX API and exports it to the application layers of the system. BSD serves as the basis for the file systems and networking facilities of Mac OS X. In addition, it provides several programming interfaces and services, including

- the process model (process IDs, signals, and so on)
- basic security policies such as user IDs and permissions
- threading support (POSIX threads)
- BSD sockets

Device-Driver Support

For development of device drivers, Darwin offers an object-oriented framework called the I/O Kit. The I/O Kit not only facilitates the creation of drivers for Mac OS X, but provides much of the infrastructure those drivers need. It is written in a restricted subset of C++. The framework, which is designed to support a range of device families, is both modular and extensible.

Graphics and Imaging

Mac OS X combines Quartz, QuickTime, and OpenGL—three of the most powerful graphics technologies available—to take the graphics capabilities of the Macintosh beyond anything seen on a desktop operating system. The two-dimensional graphics and imaging capabilities of Mac OS X are based on Quartz, an Apple technology that provides a window server and essential low-level services as well as a graphics rendering library that uses PDF (Portable Document Format) as its internal model. Integrated into this foundation is a printing architecture and other graphics libraries such as QuickDraw and QuickTime.

2.2. The architecture of Mac OS X

The central characteristic of the Mac OS X architecture is the layering of system software, with one layer having dependencies on, and interfaces with, the

layer beneath it (see Figure 2.2). Mac OS X has four distinct layers of system software (in order of dependency):

- **Application environments**. Encompasses the five application (or execution) environments: Carbon, Cocoa, Java, Classic, and BSD Commands. For developers, the first three of these environments are the most significant. Mac OS X includes development tools and runtimes for these environments.
- **Application Services**. Incorporates the system services available to all application environments that have some impact on the graphical user interface. It includes Quartz, QuickDraw, and OpenGL as well as essential system managers.
- **Core Services**. Incorporates those system services that have no effect on the graphical user interface. It includes Core Foundation, Open Transport, and certain core portions of Carbon.
- **Kernel environment**. Provides the foundation layer of Mac OS X. Its primary components are Mach and BSD, but it also includes networking protocol stacks and services, file systems, and device drivers. The kernel environment offers facilities for developing device drivers (the I/O Kit) and loadable kernel extensions, including Network Kernel Extensions (NKEs).

A common way to look at complex software is to separate out parts of that software into “layers.” Visually depicted, one layer sits on top of another, with the most fundamental layer on the bottom. This kind of diagram suggests the general interfaces and dependencies between the layers of software. The higher layers of software, which are the closest to actual application code, depend on the layer immediately under them, and that intermediate layer depends on an even lower layer.

Mac OS X is reducible to such a perspective. Figure 2.2illustrates the general structure of Mac OS X system software as interdependent layers of libraries, frameworks, and services.

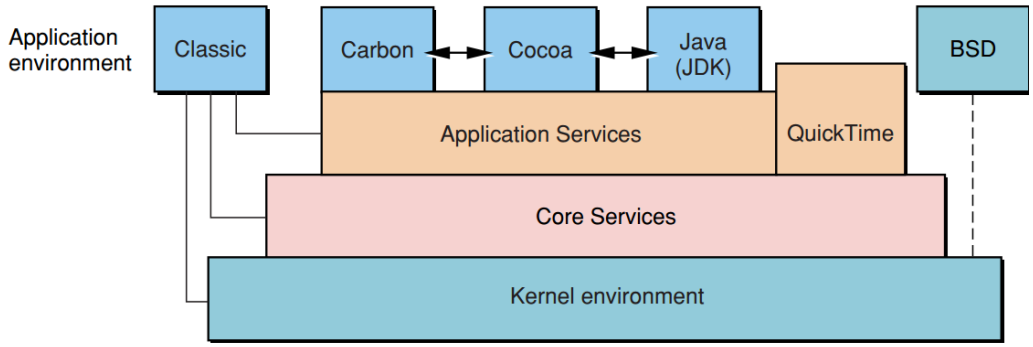

Figure 2.2. Mac OS X as layers of system software

Although this diagram does help clarify the overall architecture, there are dangers in the necessarily over-simplified view it presents. The Mac OS X services and subsystems that one application uses—and how it uses them—can be very different from those used by another application, even one of a similar type. Dependencies and interfaces at the different levels can vary from program to program depending on individual requirements and realities.

The boxes in the top row of the diagram of Figure 2.2 represent the different application (or execution) environments of Mac OS X. There are five such environments. The Classic and the BSD Commands environments are unique in the way they interact with the underlying layers of the system:

- The Classic "compatibility" environment is where users can run their Mac OS 8 or Mac OS 9 applications. Instead of sitting on top of the Application Services, the Classic environment in this diagram has lines connecting it to each layer. These connections indicate that the Classic environment is "hard-wired" into Mac OS X; it is not an environment for which developers can specifically compile code in Mac OS X. In other words, there are no public non-Carbon Mac OS 8 or Mac OS 9 APIs on a Mac OS X system that can be compiled.
- The BSD Commands environment provides a shell in which you can execute BSD programs on the command line. The standard BSD tools, utilities, and scripts are available for this environment as well as any custom ones you or third parties create. The diagram shows the BSD

Commands environment connected directly with the kernel-environment layer. Note that you can run programs on the command line that are built in non-BSD environments, such as programs based on Cocoa's Foundation framework.

The kernel environment exports BSD services to the upper layers of the system through the System library in /usr/lib (the headers are located in /usr/include). BSD commands are also available to developers; however, BSD commands might not be included in certain Mac OS X installations. Because the BSD Commands environment is a special optional environment.

Carbon, Cocoa, and Java are the three principal application environments for Mac OS X developers:

- Carbon is an adaptation of the Mac OS 9 APIs and libraries for Mac OS X. It carries over most of the prior APIs (70 percent of the functions) and includes some APIs and services specifically developed for Mac OS X.
- Cocoa is a collection of advanced object-oriented APIs for developing applications written in Java and Objective-C.
- The Java environment is for the development and deployment of 100% Pure Java and mixed-API Java applications and applets.

Directly supporting the Carbon, Cocoa, and Java environments are the layers of system software that offer services for all application environments. These layers are stacked in decreasing widths to suggest that application code can access lower layers directly—that is, without the mediation of intervening layers.

The first of these layers is the Application Services layer. It contains the graphics and windowing environment of Mac OS X, principally implemented by Quartz and QuickDraw. This environment is responsible for screen rendering, printing, event handling, and low-level window and cursor management. It also holds libraries, frameworks, and background servers useful in the implementation of graphical user interfaces.

QuickTime is an extension to the operating system that architecturally spans layers of system software. It is an interactive multimedia environment that has features and functionality common to both a graphics environment and an application environment. Figure 2.2 presents QuickTime as straddling the line between Application Services and the application environments. QuickTime requires a host application environment (or a browser) in which to execute, but the multimedia components that it offers have unique and sophisticated capabilities typically found only in application environments.

The Application Services layer sits on top of Core Services. In the Core Services layer are the common services that are not directly a part of a graphical user interface. Here you find cross-environment implementations of basic programmatic abstractions such as strings, run loops, and collections. There are also APIs in Core Services for managing processes, threads, resources, and virtual memory, and for interacting with the file system.

The kernel environment is the lowest stratum of system software, just below the Core Services layer. The kernel environment provides essential operating-system functionality to the layers above it, such as

- preemptive multitasking
- advanced virtual memory with memory protection and dynamic memory
- allocation
- symmetric multiprocessing
- multi-user access
- file systems based on VFS (Virtual File System)
- device drivers
- networking
- basic threading packages

It is a high-performance and highly modular kernel with support for dynamic loading of device drivers, networking extensions, and file systems.
The kernel environment consists of five major components:

- **Mach**. Provides the fundamental abstractions and implementations of tasks, threads, ports, virtual addressing, memory management, and intertask communication. Mach is also the part of the operating system that manages processor usage, handles scheduling, and enforces memory protection. In addition it provides timing services, synchronization primitives, and a messaging-centered infrastructure to the rest of the operating system.
- **BSD**. A version of 4.4BSD is used, among other things, to support Mach's preemptive multitasking, memory protection, dynamic memory allocation, and symmetric multiprocessing. BSD forms the basis for networking and file systems in Mac OS X. Some of the other facilities it provides or supports are process creation and management, signals, system bootstrap and shutdown, generic I/O operations, basic file operations, and handling of terminals and other devices. It also implements user and group IDs as well as the related features of resource limits and access policies for files and other resources. BSD provides many of the POSIX APIs.
- **Device drivers and the I/O Kit**. Device drivers in Mac OS X are created with the I/O Kit, a framework that offers an object-oriented programming model (based on a restrictive form of C++) to streamline the development of device drivers. The I/O Kit takes into account underlying operating-system features such as virtual memory, memory protection, and preemption and thus relieves device-driver writers from having to worry about them in their code. It is designed to be modular, reusable, and extensible. The kernel environment includes a number of ready-made device drivers.
- **Networking**. The kernel environment implements numerous native networking protocols and facilities. Some of the networking facilities and protocol stacks of Mac OS X are implemented as Network Kernel Extensions (NKEs). They can extend the networking infrastructure of the

kernel dynamically—that is, without recompiling and relinking the kernel.

- **File systems**. The kernel environment supports many different file systems and volume formats, including Mac OS Extended (HFS+), Mac OS Standard (HFS), UFS, NFS, and ISO 9660 for CD-ROMs. Mac OS Extended is the default file system, and Mac OS X typically boots and "roots" from it (that is, the kernel uses the file system on an HFS+ volume as the one to mount first). By using the Virtual File System (VFS) infrastructure, developers can write kernel extensions that add support for other file systems and extend file system functionality—adding file-level compression, for instance. VFS is a set of standard internal file-system interfaces and utilities for building such extensions.

The kernel environment, Core Services, and Application Services layers of Mac OS X are packaged as umbrella frameworks. Two of the primary Mac OS X application environments, Carbon and Cocoa, are also packaged as umbrella frameworks.

2.3. Security of Mac OS X

We provide a brief outline of a MAC OS system prior to examining the security details.

A running Mac OS system consists of an *operating system kernel* and many *processes* each running a program. A protection ring boundary isolates the MAC OS kernel from the processes. Each process has its own *address space*, that defines the memory addresses that it can access. Modern MAC OS systems define address spaces primarily in terms of the set of *memory pages* that they can access.MAC OS uses the concept of a *file* for all persistent system objects, such as secondary storage, I/O devices, network, and inter process communication. A MAC OS process is associated with an *identity*, based on the user associated with the process, and access to files is limited by the process's identity.

MAC OS security aims to protect users from each other and the system's trusted computing base (TCB) from all users. Informally, the MAC OS TCB consists of the kernel and several processes that run with the identity of the privileged user, root or superuser. These root processes provide a variety of services, including system boot, user authentication, administration, network services, etc. Both the kernel and root processes have full system access. All other processes have limited access based on their associated user's identity.

Mac OS protection system

MAC OS implements a classical protection system , not the secure protection system. MAC OS protection system consists of a p rotection state and a set of operations that enable processes to modify that state. Thus, MAC OS is a *discretionary access control* (DAC) system. However, MAC OS does have some aspects of the secure protection system. First, the MAC OS protection system defines a *transition state* that describes how processes change between protection domains. Second, the *labeling state* is largely ad hoc. Trusted services associate processes with user identities, but users can control the assignment of permissions to system resources (i.e., files). In the final analysis, these mechanisms and the discretionary protection system are insufficient to build a system that satisfies the secure operating system requirements.

Recall that a protection state describes the operations that the system's subjects can perform on that system's objects. The MAC OS protection state associates process identities (subjects) with their access to files (objects). Each MAC OS process identity consists of a *user id* (UID), a *group id* (GID), and a set of *supplementary groups*. These are used in combination to determine access as described below.

All MAC OS resources are represented as files. The protection state specifies that subjects may perform read, write, and execute operations on files, with the standard meaning of these operations. While directories are not files, they are represented as files in the MAC OS protection state, although the operations have different semantics (e.g., execute means search for a directory).

Files are also associated with an owner UID and an owner GID which conveys special privileges to processes with these identities. A process with the owner UID can modify any aspect of the protection state for this file. Processes with either the owner UID and group GID may obtain additional rights to access the file as described below.

The limited set of objects and operations enabled MAC OS designers to use a compressed access control list format called *MAC OS mode bits*, to specify the access rights of identities to files. Mode bits define the rights of three types of subjects: the file owner UID; the file group GID; and all other subjects. Using mode bits authorization is performed as follows. First, the MAC OS authorization mechanism checks whether the process identity's UID corresponds to the owner UID of the file, and if so, uses the mode bits for the owner to authorize access. If the process identity's GID or supplementary groups correspond to the file's group GID, then the mode bits for the group permissions are used. Otherwise, the permissions assigned to all others are used.

*Example***.** MAC OS mode bits are of the form {owner bits, group bits, others bits} where each element in the tuple consists of a read bit, a write bit, and an execute bit. The mode bits:

rwxr--r—

mean that a process with the same UID as the owner can read, write, or execute the file, a process with a GID or supplementary group that corresponds to the file's group can read the file, and others can also only read the file.

Suppose a set of files have the following owners , groups, and others mode bits as described below:

Name	Owner	Group	Mode Bits
foo	alice	faculty	rwxr--r--
bar	bob	students	rw-rw-r--
baz	charlie	faculty	rwxrwxrwx

Then, processes running as alice with the group faculty can read, write, or execute foo and baz, but only read bar. For bar, Alice does not match the UID

(bob), nor have the associated group (students). The process has the appropriate owner to gain all privileges for foo and the appropriate group to gain privileges to baz.

As described above, the MAC OS protection system is a discretionary access control system. Specifically, this means that a file's mode bits, owner UID, or group GID may be changed by any MAC OS processes run by the file's owner (i.e., that have the same UID as the file owner). If we trust all user processes to act in the best interests of the user, then the user's security goals can be enforced. However, this is no longer a reasonable assumption. Nowadays, users run a variety of processes, some of which may be supplied by attackers and others may be vulnerable to compromise from attackers, so the user will have no guarantee that these processes will behave consistently with the user's security goals. As a result, a secure operating system cannot use discretionary access control to enforce user security goals.

Since discretionary access control permits users to change their files owner UID and group GID in addition to the mode bits, file labeling is also discretionary. A secure protection system requires a mandatory *labeling state*, so this is another reason that MAC OS systems cannot satisfy the requirements of a secure operating system.

MAC OS processes are labeled by trusted services from a set of labels (i.e., user UIDs and group GIDs) defined by trusted administrators, and child processes inherit their process identity from their parent. This mandatory approach to labeling processes with identities would satisfy the secure protection system requirements, although it is rather inflexible.

Finally, MAC OS mode bits also include a specification for protection domain transitions, called the setuid bit. When this bit is set on a file, any process that executes the file with automatically perform a protection domain transition to the file's owner UID and group GID. For example, if a root process sets the setuid bit on a file that it owns, then any process that executes that file will run under the root UID. Since the setuid bit is a mode bit, it can be set by the file's owner, so it is

also managed in a discretionary manner. A secure protection state requires a mandatory *transition state* describe all protection domain transitions, so the use of discretionary setuid bits is insufficient.

Mac OS authorization

The MAC OS authorization mechanism controls each process's access to files and implements protection domain transitions that enable a process to change its identity. The authorization mechanism runs in the kernel, but it depends on system and user processes for determining its authorization queries and its protection state. For these and other reasons described in the MAC OS security analysis, the MAC OS authorization mechanism does not implement a reference monitor.

MAC OS authorization occurs when files are opened, and the operations allowed on the file are verified on each file access. The requesting process provides the name of the file and the operations that will be requested upon the file in the open system call. If authorized, MAC OS creates a *file descriptor* that represents the process's authorized access to perform future operations on the file. File descriptors are stored in the kernel, and only an index is returned to the process. Thus, file descriptors are a form of. User processes present their file descriptor index to the kernel when they request operations on the files that they have opened.

MAC OS authorization controls traditional file operations by mediating file open for read, write, and execute permissions. However, the use of these permissions does not always have the expected effect: (1) these permissions and their semantics do not always enable adequate control and (2) some objects are not represented as files, so they are unmediated. If a user has read access to a file, this is sufficient to perform a wide-variety of operations on the file besides reading. For example, simply via possession of a file descriptor, a user process can perform any ad hoc command on the file using the system calls ioctl or fcntl, as well as read and modify file metadata. Further, MAC OS does not mediate all security-sensitive objects, such as network communications. Host firewalls provide some control of

network communication, but they do not restrict network communication by process identity.

The MAC OS authorization mechanism depends on user-level authentication services, such as login and sshd, to determine the process identity (i.e., UID, GID, and supplementary groups). When a user logs in to a system, her processes are assigned her login identity. All subsequent processes created in this login session inherit this identity unless there is a domain transition (see below). Such user-level services also need root privileges in order to change the identity of a process, so they run with this special UID. However, several MAC OS services need to run as root in order to have the privileges necessary to perform their tasks. These privileges include the ability to change process identity, access system files and directories, change file permissions, etc. Some of these services are critical to the correct operation of MAC OS authorization, such as sshd and passwd, but others are not, such as inetd and ftp. However, a MAC OS system's trusted computing base must include all root processes, thus risking compromise of security critical services and the kernel itself.

MAC OS protection domain transitions are performed by the setuid mechanism. setuid is used in two ways: a root process can invoke the setuid system call to change the UID of a process and a file can have its setuid mode bit set, such that whenever it is executed its identity is set to the owner of the file. In the first case, a privileged process, such as login or sshd, can change the identity of a process. For example, when a user logs in, the login program must change the process identity of the user's first process, her shell, to the user to ensure correct access control. In the second case, the use of the setuid bit on a file is typically used to permit a lower privileged entity to execute a higher privileged program, almost always as root. For example, when a user wishes to change her password, she uses the passwd program. Since the passwd program modifies the password file, it must be privileged, so a process running with the user's identity could not change the password file. The setuid bit on the root-owned, passwd executable's file is set, so when any user executes passwd, the resultant process identity

transitions to root. While the identity transition does not impact the user's other processes, the writers of the passwd program must be careful not to allow the program to be tricked into allowing the user to control how passwd uses its additional privileges.

MAC OS also has a couple of mechanisms that enable a user to run a process with a reduced set of permissions. Unfortunately, these mechanisms are difficult to use correctly, are only available to root processes, and can only implement modest restrictions. First, MAC OS systems have a special principal nobody that owns no files and belongs to no groups. Therefore, a process's permissions can be restricted by running as nobody since it never has owner or group privileges. Unfortunately, nobody, like all subjects, has others privileges. Also, since only root can do a setuid only a superuser process can change the process identity to nobody. Second, MAC OS chroot can be used to limit a process to a subtree of the file system. Thus, the process is limited to only its rights to files within that subtree. Unfortunately, a chroot environment must be setup carefully to prevent the process from escaping the limited domain. For example, if an attacker can create /etc/passwd and /etc/shadow files in the subtree, she can add an entry for root, login as this root, and escape the chroot environment (e.g., using root access to kernel memory). Also, a chroot environment can only be setup by a root process, so it is not usable to regular system users. In practice, neither of these approaches has proven to be an effective way to limit process permissions.

Summary

As a result of the discretionary protection system, the size of the system TCB, and these types of vulnerabilities, converting a Mac OS system to a secure operating system is a significant challenge. Ensuring that TCB processes protect themselves, and thus protect a reference monitor from tampering, is a complex undertaking as untrusted processes can control how TCB processes are invoked and provide inputs in multiple ways: network, environment, and arguments. Further, untrusted processes may use system interfaces to manipulate any shared

resources and may even change the binding between object name and the actual object. We will discuss the types of changes necessary to convert an ordinary Mac OS system to a system that aims to satisfy the secure operating system definition, so we will see that several fundamental changes are necessary to overcome these problems. Even then, the complexity of Mac OS systems and their trusted computing base makes satisfying the tamperproof and verifiability requirements of the reference monitor concept very difficult.

3. THE EVALUATION OF PROTECTION LEVEL OF MAC OS X

3.1. Requirements to security of Mac OS X

Operating systems are the software that provides access to the various hardware resources (e.g., CPU, memory, and devices). Any program that is run on a computer system has instructions executed by that computer's CPU, but these programs may also require the use of other peripheral resources of these complex systems. Consider a program that allows a user to enter her password. The operating system provides access to the disk device on which the program is stored, access to device memory to load the program so that it may be executed, the display device to show the user how to enter her password, and keyboard and mouse devices for the user to enter her password. Of course, there are now a multitude of such devices that can be used seamlessly, for the most part, thanks to the function of operating systems.

Operating systems run programs in *processes.* The challenge for an operating system developer is to permit multiple concurrently executing processes to use these resources in a manner that preserves the independence ofthese processes while providing fair sharing of these resources. Originally, operating systems only permitted one process to be run at a time (e.g., *batch systems),* but as early as 1960, it became apparent that computer productivity would be greatly enhanced by being able to run multiple processes concurrently. By *concurrently,* we mean that while only one process uses a computer's CPU at a time, multiple other processes may be in various states of execution at the same time, and the operating system must ensure that these executions are performed effectively. For example, while the computer waits for a user to enter her password, other processes may be run and access system devices as well, such as the network. These systems were originally called *timesharing systems,* but they are our default operating systems today.

To build any successful operating system, we identify three major tasks. First, the operating system must provide various mechanisms that enable high

performance use of computer resources. Operating systems must provide efficient *resource mechanisms*, such as file systems, memory management systems, network protocol stacks, etc., that define how processes use the hardware resources. Second, it is the operating system's responsibility to switch among the processes fairly, such that the user experiences good performance from each process in concert with access to the computer's devices. This second problem is one of *scheduling* access to computer resources. Third, access to resources should be controlled, such that one process cannot inadvertently or maliciously impact the execution of another. This third problem is the problem of ensuring the *security* of all processes run on the system.

Ensuring the secure execution of all processes depends on the correct implementation of resource and scheduling mechanisms. First, any correct resource mechanism must provide boundaries between its objects and ensure that its operations do not interfere with one another. For example, a file system must not allow a process request to access one file to overwrite the disk space allocated to another file. Also, file systems must ensure that one write operation is not impacted by the data being read or written in another operation. Second, scheduling mechanisms must ensure availability of resources to processes to prevent denial of service attacks. For example, the algorithms applied by scheduling mechanisms must ensure that all processes are eventually scheduled for execution. These requirements are fundamental to operating system mechanisms, and are assumed to be provided in the context of this book. The scope of this book covers the misuse of these mechanisms to inadvertently or, especially, maliciously impact the execution of another process.

Security becomes an issue because processes in modern computer systems interact in a variety of ways, and the sharing of data among users is a fundamental use of computer systems. First, the output of one process may be used by other processes. For example, a programmer uses an editor program to write a computer program's source code, compilers and linkers to transform the program code into a form in which it can be executed, and debuggers to view the executing processes

image to find errors in source code. In addition, a major use of computer systems is to share information with other users. With the ubiquity of Internet-scale sharing mechanisms, such as e-mail, the web, and instant messaging, users may share anything with anyone in the world. Unfortunately, lots of people, or at least lots of email addresses, web sites, and network requests, want to share stuff with you that aims to circumvent operating system security mechanisms and cause your computer to share additional, unexpected resources. The ease with which malware can be conveyed and the variety of ways that users and their processes may be tricked into running malware present modern operating system developers with significant challenges in ensuring the security of their system's execution.

The challenge in developing operating systems security is to design security mechanisms that protect process execution and their generated data in an environment with such complex interactions. As we will see, formal security mechanisms that enforce provable security goals have been defined, but these mechanisms do not account or only partially account for the complexity of practical systems. As such, the current state of operating systems security takes two forms: constrained systems that can enforce security goals with a high degree of assurance and general-purpose systems that can enforce limited security goals with a low to medium degree of assurance. First, several systems have been developed over the years that have been carefully crafted to ensure correct (i.e., within some low tolerance for bugs) enforcement of specific security goals. These systems generally support few applications, and these applications often have limited functionality and lower performance requirements. That is, in these systems, security is the top priority, and this focus enables the system developers to write software that approaches the ideal of the formal security mechanisms mentioned above. Second, the computing community at large has focused on function and flexibility, resulting in general-purpose, extensible systems that are very difficult to secure. Such systems are crafted to simplify development and deployment while achieving high performance, and their applications are built to be feature-rich and easy to use. Such systems present several challenges to security

practitioners, such as insecure interfaces, dependence of security on arbitrary software, complex interaction with untrusted parties anywhere in the world, etc. But, these systems have defined how the user community works with computers. As a result, the security community faces a difficult task for ensuring security goals in such an environment.

However, recent advances are improving both the utility of the constrained systems and the security of the general-purpose systems. We are encouraged by this movement, which is motivated by the general need for security in all systems, and this book aims to capture many of the efforts in building security into operating systems, both constrained and general-purpose systems, with the aim of enabling broader deployment and use of security function in future operating systems.

The ideal goal of operating system security is the development of a secure operating system. *A secure operating system provides security mechanisms that ensure that the system's security goals are enforced despite the threats faced by the system.* These security mechanisms are designed to provide such a guarantee in the context of the resource and scheduling mechanisms. Security goals define the requirements of secure operation for a system for any processes that it may execute. The security mechanisms must ensure these goals regardless of the possible ways that the system maybe misused (i.e., is threatened) by attackers.

The term "secure operating system" is both considered an ideal and an oxymoron. Systems that provide a high degree of assurance in enforcement have been called secure systems, or even more frequently "trusted" systems. However, it is also true that no system of modern complexity is completely secure. The difficulty of preventing errors in programming and the challenges of trying to remove such errors means that no system as complex as an operating system can be completely secure.

Nonetheless, we believe that studying how to build an ideal secure operating system to be useful in assessing operating systems security. we develop a definition of *secure operating system* that we will use to assess several operating

systems security approaches and specific implementations of those approaches. While no implementation completely satisfies this ideal definition, its use identifies the challenges in implementing operating systems that satisfy this ideal in practice. The aim is multi-fold. First, we want to understand the basic strengths of common security approaches. Second, we want to discover the challenges inherent to each of these approaches. These challenges often result in difficult choices in practical application. Third, we want to study the application of these approaches in practical environments to evaluate the effectiveness ofthese approaches to satisfy the ideal in practice. While it appears impractical to build an operating system that satisfies the ideal definition, we hope that studying these systems and their security approaches against the ideal will provide insights that enable the development of more effective security mechanisms in the future.

To return to the general definition of a secure operating system from the beginning of this section, we examine the general requirements of a secure operating system. To build any secure system requires that we consider how the system achieves its *security goals* under a set of threats (i.e., a *threat model)* and given a set of software, including the security mechanisms, that must be trusted [2] (i.e., a *trust model).*

A security goal defines the operations that can be executed by a system while still preventing unauthorized access. It should be defined at a high-level of abstraction, not unlike the way that an algorithm's worst-case complexity prescribes the set of implementations that satisfy that requirement. A security goal defines a requirement that the system's design can satisfy (e.g., the way pseudocode can be proven to fulfill the complexity requirement) and that a correct implementation must fulfill (e.g., the way that an implementation can be proven experimentally to observe the complexity).

Security goals describe how the system implements accesses to system resources that satisfy the following: *secrecy, integrity,* and *availability.* A system access is traditionally stated in terms of which *subjects* (e.g., processes and users) can perform which *operations* (e.g., read and write) on which *objects* (e.g., files

and sockets). Secrecy requirements limit the objects that individual subjects can *read* because objects may contain secrets that not all subjects are permitted to know. Integrity requirements limit the objects that subjects can *write* because objects may contain information that other subjects *depend on* for their correct operation. Some subjects may not be trusted to modify those objects. Availability requirements limit the system resources (e.g., storage and CPU) that subjects may *consume* because they may exhaust these resources. Much of the focus in secure operating systems is on secrecy and integrity requirements, although availability may indirectly impact these goals as well.

The security community has identified a variety of different security goals. Some security goals are defined in terms of security requirements (i.e., secrecy and integrity), but others are defined in terms of function, in particular ways to limit function to improve security. An example of a goal defined in terms of security requirements is the *simple-security property* of the Bell-LaPadula model. This goal states that a process cannot read an object whose secrecy classification is higher than the process's. This goal limits operations based on a security requirement, secrecy. An example of an functional security goal is the *principle of least privilege*, which limits a process to only the set of operations necessary for its execution. This goal is functional because it does not ensure that the secrecy and/or integrity of a system is enforced, but it encourages functional restrictions that may prevent some attacks. However, we cannot prove the absence of a vulnerability using functional security goals.

The task of the secure operating system developer is to define security goals for which the security of the system can be verified, so functional goals are insufficient. On the other hand, secrecy and integrity goals prevent function in favor of security, so they may be too restrictive for some production software. In the past, operating systems that enforced secrecy and integrity goals (i.e., the constrained systems above) were not widely used because they precluded the execution of too many applications (or simply lacked popular applications). Emerging technology, such as virtual machine technology, enables multiple,

commercial software systems to be run in an isolated manner on the same hardware. Thus, software that used to be run on the same system can be run in separate, isolated virtual systems. It remains to be seen whether such isolation can be leveraged to improve system security effectively. Also, several general-purpose operating systems are now capable of expressing and enforcing security goals. Whether these general-purpose systems will be capable of implementing security goals or providing sufficient assurance for enforcing such goals is unclear. However, in either case, security goals must be defined and a practical approach for enforcing such goals, that enables the execution of most popular software in reasonable ways, must be identified.

A system's *trust model* defines the set of software and data upon which the system depends for correct enforcement of system security goals. For an operating system, its trust model is synonymous with the system's *trusted computing base* (TCB).

Ideally, a system TCB should consist of the minimal amount of software necessary to enforce the security goals correctly. The software that must be trusted includes the software that defines the security goals and the software that enforces the security goals (i.e., the operating system's security mechanism). Further, software that bootstraps this software must also be trusted. Thus, an ideal TCB would consist of a bootstrapping mechanism that enables the security goals to be loaded and subsequently enforced for lifetime of the system.

In practice, a system TCB consists of a wide variety of software. Fundamentally, the enforcement mechanism is run within the operating system. As there are no protection boundaries between operating system functions (i.e., in the typical case of a monolithic operating system), the enforcement mechanism must trust all the operating system code, so it is part of the TCB.

Further, a variety ofother software running outside the operating system must also be trusted. For example, the operating system depends on a variety of programs to authenticate the identity of users (e.g., login and SSH). Such programs must be trusted because correct enforcement of security goals depends on correct

identification of users. Also, there are several services that the system must trust to ensure correct enforcement of security goals. For example, windowing systems, such as the Mac OS X, perform operations on behalf of all processes running on the operating system, and these systems provide mechanisms for sharing that may violate the system's security goals (e.g., cut-and-paste from one application to another). As a result, the Mac OS X and a variety of other software must be added to the system's TCB.

The secure operating system developer must prove that their systems have a viable trust model. This requires that: the system TCB must mediate all security-sensitive operations; verification of the correctness of the TCB software and its data; and verification that the software's execution cannot be tampered by processes outside the TCB. First, identifying the TCB software itself is a nontrivial task for reasons discussed above. Second, verifying the correctness of TCB software is a complex task. For general-purpose systems, the amount of TCB software outside the operating system is greater than the operating system software that is impractical to verify formally. The level of trust in TCB software can vary from software that is formally-verified (partially), fully-tested, and reviewed to that which the user community trusts to perform its appointed tasks. While the former is greatly preferred, the latter is often the case. Third, the system must protect the TCB software and its data from modification by processes outside the TCB. That is, the integrity of the TCB must be protected from the threats to the system, described below. Otherwise, this software can be tampered, and is no longer trustworthy.

A *threat model* defines a set of operations that an *attacker* may use to *compromise* a system. In this threat model, we assume a powerful attacker who is capable of injecting operations from the network and may be in control of some ofthe running software on the system (i.e., outside the trusted computing base). Further, we presume that the attacker is actively working to violate the system security goals. If an attacker is able to find a vulnerability in the system that provides access to secret information (i.e., violate secrecy goals) or permits the

modification of information that subjects depend on (i.e., violate integrity goals), then the attacker is said to have compromised the system.

Since the attacker is actively working to violate the system security goals, we must assume that the attacker may try any and all operations that are permitted to the attacker. For example, if an attacker can only access the system via the network, then the attacker may try to send any operation to any processes that provide network access. Further, if an attacker is in control of a process running on the system, then the attacker will try any means available to that process to compromise system security goals.

This threat model exposes a fundamental weakness in commercial operating systems (e.g., UNIX and Windows); they assume that all software running on behalf of a subject is trusted by that subject. For example, a subject may run a word processor and an email client, and in commercial systems these processes are trusted to behave as the user would. However, in this threat model, both of these processes may actually be under the control of an attacker (e.g., via a document macro virus or via a malicious script or email attachment). Thus, a secure operating system cannot trust processes outside of the TCB to behave as expected. While this may seem obvious, commercial systems trust any user process to manage the access of that user's data (e.g., to change access rights to a user's files via chmod in a UNIX system). This can result in the leakage of that user's secrets and the modification of data that the user depends on.

The task of a secure operating system developer is to protect the TCB from the types of threats described above. Protecting the TCB ensures that the system security goals will always be enforced regardless of the behavior of user processes. Since user processes are untrusted, we cannot depend on them, but we can protect them from threats. For example, secure operating system can prevent a user process with access to secret data from leaking that data, by limiting the interactions of that process. However, protecting the TCB is more difficult because it interacts with a variety of untrusted processes. A secure operating system developer must identify such threats, assess their impact on system security, and

provide effective countermeasures for such threats. For example, a trusted computing base component that processes network requests must identify where such untrusted requests are received from the network, determine how such threats can impact the component's behavior, and provide countermeasures, such as limiting the possible commands and inputs, to protect the component. The secure operating system developer must ensure that all the components of the trusted computing base prevent such threats correctly.

3.2. Security analysis of Mac OS X

Mac OS X has broad appeal to both end-users and administrators alike. It melds the familiar features and robustness of UNIX with the user-friendliness expected from a Macintosh OS into a cohesive package. Administrators in heterogeneous environments that include UNIX and Linux systems will welcome Mac OS X to the stable of systems they support. They will appreciate its versatility, built-in security mechanisms, and remote administration capabilities. End-users will value the stable environment provided by Mac OS X's protected memory and pre-emptive multitasking kernel. Apple's latest offering also has the distinction of being the only UNIX flavor that has a native version of Microsoft Office. This feature alone will help Mac OS X to gain a more widespread presence than any other flavor of UNIX.

Because of its broad appeal and its potential to become the most widespread BSD distribution on the desktop, it is imperative that administrators understand the risks Mac OS X introduces to their networks. As security-conscious administrators know, a network is only as secure as its weakest link. Apple's new operating system presents security concerns both common to most UNIX- like operating systems and peculiar to Mac OS X. This paper addresses security concerns regarding choice of filesystem, physical access, NetInfo directories, password authentication, the internet superserver and other UNIX daemons, and root privilege mechanisms. I will also address strengths such as Mac OS X's built-in firewall, its semi-automated software update feature, its support of popular

commercial anti-virus software, and its ability to run powerful UNIX security and security auditing tools.

I have devoted a considerable portion of my research to NetInfo, Mac OS X's directory system. This topic held much interest for me because I was wholly unfamiliar with the system. I found many sources that described NetInfo's primary vulnerability: Unprivileged users and processes can extract sensitive information from a NetInfo directory. I read of a possible solution that involved moving sensitive information to the traditional, protected locations used by other UNIX distributions. I did not, however, find resources that described the effectiveness of this solution or the problems that may arise by altering Apple's intended NetInfo configuration scheme. This lack of information led me to perform my own experiments and I present my results in detail below (see NetInfo and Disclosure of Sensitive Information).

The earliest version of the Classic environment supported by Mac OS X is Mac OS 9.1. For the sake of brevity I will use the term "Mac OS 9.x" to refer to Mac OS 9.1 and Mac OS 9.2.x, and I will use "Mac OS X" and "X" interchangeably. I will also use the convention that GUI tools, like *NetInfo Manager*, will be italicized. I will precede command-line examples with a ">" prompt to indicate that these are commands that should be entered at a shell prompt. Additionally, I will use the term "administrative user" to refer to any user belonging to the group "admin," but will use the term "administrators" to refer to the target audience of this paper—those people responsible for maintenance and security of systems on their networks. The two groups may overlap, but I make the distinction here for clarity.

I have devoted a considerable portion of my research to NetInfo, Mac OS X's directory system. This topic held much interest for me because I was wholly unfamiliar with the system. I found many sources that described NetInfo's primary vulnerability: Unprivileged users and processes can extract sensitive information from a NetInfo directory. I read of a possible solution that involved moving sensitive information to the traditional, protected locations used by other UNIX

distributions. I did not, however, find resources that described the effectiveness of this solution or the problems that may arise by altering Apple's intended NetInfo configuration scheme. This lack of information led me to perform my own experiments and I present my results in detai l below (see NetInfo and Disclosure of Sensitive Information).

The earliest version of the Classic environment supported by Mac OS X is Mac OS 9.1. For the sake of brevity I will use the term "Mac OS 9.x" to refer to Mac OS 9.1 and Mac OS 9.2.x, and I will use "Mac OS X" and "X" interchangeably. I will also use the convention that GUI tools, like *NetInfo Manager,* will be italicized. I will precede command-line examples with a ">" prompt to indicate that these are commands that should be entered at a shell prompt. Additionally, I will use the term "administrative user" to refer to any user belonging to the group "admin," but will use the term "administrators" to refer to the target audience of this paper—those people responsible for maintenance and security of systems on their networks. The two groups may overlap, but I make the distinction here for clarity.

Filesystem Considerations

Mac OS X supports installation on disk partitions formatted using UFS or HFS+ filesystems. There are advantages to installing Mac OS X on a UFS volume. These security advantages outweigh the minor pitfalls associated with using the UFS filesystem.

UFS is case-sensitive like most UNIX filesystems whereas HFS+ is case-preserving, but case-insensitive. The case-insensitive nature of HFS+ creates vulnerabilities. With regard to security, programmers invariably make assumptions. Conclusions drawn from these assumptions become invalid when the assumptions no longer hold true. A well-known illustration of this point with respect to Mac OS X is the use of Apache to serve files on HFS+ volumes. Apache programmers assumed served filesystems would be case-sensitive and this assumption created vulnerabilities when Apple shipped Mac OS X, with Apache, on HFS+ volumes. A fix has been available for some time, but Apache is certainly

not the only UNIX program that has made this assumption. Additional vulnerabilities associated with using UNIX-derived tools on Mac OS X can be avoided by limiting their use to UFS filesystems.

Both filesystems support UNIX metadata (owner and mode bits) necessary for filesystem discretionary access control (DAC) mechanisms, which are common to all flavors of UNIX. On a UFS filesystem, this metadata is stored in a file's inode. On a HFS+ filesystem, this metadata is stored with the file data because HFS+ does not use inodes. Although HFS+ supports DAC metadata, Mac OS 9.x neither creates this metadata nor respects DAC metadata created by Mac OS X. A disadvantage of installing X on a HFS+ volume is that booting into Mac OS 9 X will circumvent DAC mechanisms and allow a user unrestricted access to your Mac OS X volume. Mac OS 9.x cannot, however, mount or read UFS volumes. This inability to use UFS volumes may be inconvenient, but putting a Mac OS X installation on a UFS volume reduces the vulnerabilities posed by physical access to the computer.

There are several caveats regarding the installation of Mac OS X on a UFS volume. If X is installed on a UFS volume, the system volume will be named "/" instead of the default "Mac OS X" and any customization of the volume name will be lost upon restart. Having the root of the filesystem named "/" is at worst an annoyance and at best a familiar feature to UNIX users. Perhaps the only genuine problem will persist only as long as Classic applications continue to be used: The type/creator code scheme used to associate documents with applications in Classic will not function if Mac OS X is installed on a UFS volume. If a user of Mac OS X relies on Classic applications, the convenience of launching these applications with a simple double-click on a document in Finder may outweigh the benefit of improved security that the UFS filesystem offers. Other issues with using Mac OS X on UFS seem trivial. These issues include an "Open Enclosing Folder" bug in Sherlock and an issue with the Classic environment not starting without some manual modifications.

Airport, Apple's name for 802.11b wireless Ethernet, presents its own security risks: ease of network sniffing and decryption of traffic by attackers even if WEP is used. As of Mac OS X 10.1, however, Airport now functions when X is installed on a UFS volume. Airport functionality is the only problem regarding the use of UFS that Apple has addressed to date, but the outstanding UFS issues have minimal impact upon a Mac OS X system. Overall, UFS filesystems presents fewer vulnerabilities than HFS+ filesystems. Unless the use of type/creator codes is imperative, my recommendation would be to ensure that Mac OS X is installed on a UFS volume.

Physical Access Concerns

Physical access to a Macintosh running Mac OS X presents a number of security concerns. Just as with other flavors of UNIX, physical access to the machine provides the ability to bypass existing security mechanisms by booting from a device other than the default boot device (e.g. installation CDs or alternate boot disks, either internal or external). Booting from installation CDs, especially, poses greater risks for Mac OS X systems than it does for other commercial UNIX flavors because the installation CDs are easier to obtain. Every new Macintosh ships with installation CDs, whereas, in my own experience, Sun SPARCstations and SGI IRIX workstations most often do not ship with installation CDs. I cannot speak for IBM AIX or HP-UX systems, and it is true that both Solaris can easily be downloaded from Sun's website and any distribution of Linux can be downloaded from numerous mirror sites, but Mac OS X installation CD s may be more prolific due to likely higher-volume sales of Macintoshes than other UNIX systems. Due to the ease of obtaining and booting installation CDs or booting from alternate boot volumes and bypassing filesystem permissions, it is doubly important that administrators of Mac OS X systems understand the methods available to boot alternate startup volumes and what measures can be taken to prevent unauthorized use of these features.

Administrative users can boot from alternate locations by selecting an alternate System folder in the *Startup Disk* preference pane in *System Preferences,* by

holding down the *option* key during startup, or by holding down the *c* key to boot from an installation CD. If the system is booted into Mac OS 9.x, filesystem permissions on HFS+ volumes can be circumvented, allowing the equivalent of root-level access to those volumes. Booting from alternate Mac OS X installation locations circumvents filesystem permissions for both HFS+ *and* UFS volumes.
If an attacker has installed X on an external drive and can boot from it, he can authenticate against his own root or administrative user password hash on the external drive instead of the hashes stored in the default boot device. Installing X on a UFS volume obviously imparts no resistance to this method of attack.

Apple has also provided a method by which a user may reset any user password on a Mac OS X system. This is accomplished by booting from a Mac OS X CD and selecting "Reset Password" from the *Installer* menu. Apple considers this a feature. It will certainly be useful in a home setting where an administrative user may not understand the importance of remembering passwords, but it presents a risk of which any administrator should be aware.

"Target Disk Mode" also enables booting from an alternate volume. Mac OS X systems that have built-in FireWire ports can be started up in Target Disk Mode by holding down the *t* key upon startup. Connecting another Macintosh via FireWire cable to the system booted in Target Disk Mode will allow the mounting of its volumes. If the host computer is running Mac OS 9.x, it will be able to mount HFS+ volumes on the target computer. If the host computer is running Mac OS X, it will be able to mount UFS *and* HFS+ volumes. Either way, the host computer will potentially gain root-level access to any volumes it can mount.

Another method of booting a Mac OS X system is single user mode. One may enter single user mode by simply holding down the *command-s* key sequence during system startup. The risk here is that single user mode requires no authentication by default and imparts root-level access to the system.

The most apparent method to eliminate these risks associated with physical access to a Mac OS X system is to change the "security-mode" variable in the system's Open Firmware. This setting is supported by Apple Open Firmware 4.1.7

and later. Supported values for this setting are "none" (the default), "command," or "full." The effects of these values of the "security-mode" variable, at the Open Firmware prompt, are described clearly by Code Samurai in a SecureMac.com article:

The "command" mode just restricts the commands that may be executed to "go and "boot." Additionally, under the "command" mode, the "boot" command may not have any arguments—that is, it will only boot the device specified in the boot device [sic] variable; no other command may be entered or any settings changed unless the password is supplied. Moreover, this password protection feature also applies to booting up with the option key held down (which allows you to choose from available bootable volumes...).Finally, in "full" mode, the machine is completely prohibited from booting until the password is entered .

Apple provides a GUI utility called, appropriately enough, "Open Firmware Password" to set the Open Firmware security-mode variable to "command" and create an Open Firmware password. Once these settings are enabled and a password is set, (in addition to the Open Firmware command restrictions outlined above) keys that affect normal startup are disabled. An Apple Knowledge Base document provides details:

When turned off, Open Firmware Password Protection:

- blocks the ability to use the "C" key to start up from a CD- ROM disc.
- blocks the ability to use the "N" key to start up from a NetBoot server.
- blocks the ability to use the "T" key to start up in Target Disk Mode (on computers that offer this feature).
- blocks the ability to start up in Verbose mode by pressing the Command-V key combination during startup.
- block [sic] the ability to start up a system in Single-user mode by depressing the Command-S key combination during startup.
- blocks a reset of Parameter RAM (PRAM) by pressing the Command-Option-P-R key combination during startup.

- requires the password to use the Startup Manager, accessed by pressing the Option key during startup...
- requires the password to enter commands after starting up in Open Firmware, which is done by depressing the Command- Option-O-F key combination during startup.

To enable these keys again the Open Firmware Password application must be used to reset the security-mode variable to "none.” The password can be reset and changed 1) by any user of the admin group, 2) by starting up the computer from a Mac OS 9.x System Folder, or 3) if one has access to the internal hardware of the Macintosh. If the first method poses a risk, then administrators should verify that all users belonging to the admin group require such privilege and shouldconsiderusing thesudoutilitytoallow finer-grainedcontrolof administrative privileges than the admin group scheme allows (see Authorized Root Privilege Mechanisms, below). The Open Firmware password itself will prevent all but one method (the *Startup Disk* preference pane) of booting Mac OS 9.x, so method two should pose no risk. If there is a threat associated with the vulnerability of physical access to the internal hardware, an administrator should lock the case of the Macintosh.

It is important to note that Apple neither supports nor endorses the use of these Open Firmware security measures on versions of Mac OS X earlier than 10.1 or when used with third-party software utilities. Improperly changing Open Firmware settings may cause damage that only Apple can repair and these repairs may not be covered by Apple’s warranty. Good examples of potential harm are reports of permanent Open Firmware corruption if the Open Firmware password is not disabled before performing a firmware update.

Leaving a system unattended while logged-in as a user with administrator privileges or with an open shell that has administrator or root privileges is against recommended practices on any flavor of UNIX. All users should password protect their screen saver and activate i t when they step away from a Mac OS X system. This will prevent passersby from tampering with the system. One may enable this

effect by clicking the “Use my account password” in the “Activation” tab of the *Screen Saver* panel in *System Preferences*. One should also select an appropriately short delay for screen saver activation using the slider here and create a hot-corner for immediate activation of the screen saver in the “Hot Corners” tab.

3.3. Recommendations to ensure protection of Mac OS X

Netlnfo is intended to function as a hierarchical, distributed directory system and it performs this function well. It was not designed, however, to be secure.

Netlnfo provides information to Netlnfo domain clients in a fashion similar to Sun’s NlS system. It also presents similar vulnerabilities. As with NIS, an unprivileged user can obtain any directory information from the Netlnfo domain to which a Mac OS X system is bound (either from a Netlnfo server or from the local database). Administrators can configure Netlnfo tools to consult traditional UNlX "flat files” rather than Netlnfo directories when C library functions make directory queries. This presents the possibility to restrict viewing of sensitive data by using filesystem DAC mechanisms. This section will explore this alternative, present a procedure for implementing it, and explain the consequences of altering Apple’s default configuration. l will also describe how and when Mac OS X backs up Netlnfo, touch upon the existing "shadow property” security mechanism of Netlnfo,andpresentchangesto restrictaccessto Netlnfo’scommand-line utilities.

The simplest route for an administrator who wishes to maintain flat files may be to use these third-party NetInfo tools and the *User* preference pane to modify the system’s configuration and have cron update his flat files regularly. Perhaps the dearth of information that I have found on the Internet regarding the above lookupd changes and their consequences is a sign that Mac OS X administrators and developers are throwing their hands up over the issue of NetInfo security and simply decided to live with it and its deficiencies rather than deal with the inconvenience of moving data from NetInfo and maintaining flat files.

Other issues with NetInfo regard the differences between its Mac OS X and Mac OS X Server implementations. Mac OS X Server uses valueless NetInfo properties called “_shadow_*<propertyname>"* that are intended to hide the value of “ *propertyname”* from unprivileged NetInfo requests for such directory information. The Apple Knowledge Base only explicitly describes this shadowing effect for Mac OS X Server 1.x, but the *User* preference pane of Mac OS X, nevertheless, creates “_shadow_passwd" properties for new users. This makes sense because Mac OS X and Mac OS X Server must certainly share much source code. On Mac OS X, though, this shadow property does not prevent unprivilegedrequestsfrom retrievingthe value ofthe“passwd"property (the users’ password hash). It would seem that Apple has removed this shadowing behavior in Mac OS X, but did not remove its Mac OS X Server vestiges. Does Mac OS X Server’s NetInfo property shadowing work well? From what I have read, it can be easily circumvented, and a false sense of security is often worse than none at all. The removal of this shadow property functionality is no great loss to Mac OS X administrators.

So is the lookupd FFAgent the answer to hiding user password hashes and other sensitive directory information? Unfortunately, it is not. After reading about the "_shadow_passwd” functionality in Mac OS X Server, l wanted to verify that Mac OS X ignored this property. Apple describes the intended Mac OS X Server behavior of the "_shadow_passwd” property in Knowledge Base article 60112:
When an executable without special privileges uses BSD-level APls such as getpwent, getpwnam, and getpwuid, an asterisk ("*") is returned for the passwd field rather than the actual protected password.

My goal was to determine the Mac OS X behavior of unprivileged getpwuid() calls when lookupd uses the NlAgent and the FFAgent. l expected if the NlAgent were used that getpwuid() would return a valid hash, regardless of the existence of the "_shadow_passwd” property for a user. l also expected that if the FFAgent were used that getpwuid() would return an asterisk instead of a valid hash because unprivileged calls would consult the passwd file rather than the master.passwd file, which they would not be able to read.

To test my hypothesis, l configured lookupd to search using only the NlAgent. l wrote and compiled this simple bit of code:

```
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>
int main (int argc, char* argv) {
struct passwd *p;
p = getpwuid(0);
printf("root hash is %s\n",p->pw_passwd); p = getpwuid(505);
printf("test hash is %s\n",p->pw_passwd);  }
```

I then configured lookupd to search for user information using only the FFAgent and ran the above test program, again as an unprivileged user. The results were surprising. The program printed the valid hashes for both users as found in /etc/master.passwd. I have tested this on two systems because I was skeptical.

I encourage Mac OS X administrators to verify these results for themselves. My findings are that any Mac OS X user can determine all user password hashes regardless of whether the hashes exist in a NetInfo directory or the mode 0600 shadow file, /etc/master.passwd.

NetInfoConclusionsand Recommendations

Where does this leave the frustrated administrator? We can lessen our risk by denying access to the NetInfo utilities and our NetInfo database(s) to unprivileged users. This should be done regardless of whether an administrator decides to use flat files or NetInfo.

```
#!/bin/sh
Change permissions for NI CLI utilities
chmod go-rwx /usr/bin/nicl
chmod go-rwx /usr/bin/nireport
chmod go-rwx /usr/bin/niutil
chmod go-rwx /usr/bin/nigrep
chmod go-rwx /usr/bin/nifind
```

chmod go-rwx /usr/bin/nidump

chmod go-rwx /usr/bin/niload

Change permissions for Aqua NetInfo Manager chmod o-rwx \

/Applications/Utilities/NetInfo\ Manager.app/Contents/MacOS/NetInfo\ Manager

Change permissions for NetInfo Databases

chmod go-rwx /var/db/netinfo/local.nidb

chmod go-rwx 'find /var/db/netinfo -name '*.nidb"

Change permissions for NetInfo Backup Directory

chmod go-rwx /var/backups/

These changes will prevent unprivileged users from using the NetInfo tools, but they will not prevent them from copying their own versions of the tools from another Macintosh or writing C programs that use the getpw*() system calls. I encourage administrators, nonetheless, to make these changes.

NetInfo was designed as a directory system, not a secure authentication system, and was created before much emphasis was placed on computer security. Its behavior reflects these roots. NetInfo and its associated tools will, on a default installation on X, allow any user to obtain sensitive system configuration and user data. Altering the search order of *lookupd* may allow UNlX administrators to more easily maintain consistent configurations across their UNlX platforms, but at this time it does not impart any additional security to Mac OS X. lf and when Apple fixes the behavior of lookupd and query functions like getpwuid() and its ilk, this alternative may prove more secure than using Netlnfo for password authentication. For now, the only useful measure administrators can take is to restrict access to the Netlnfo command-line utilities using filesystem DAC mechanisms.

Password Concerns and Strength Testing

Perhaps of more concern than the disclosure of password hashes is whether or not user passwords are strong and resistant to dictionary attacks. The Mac OS X passwd command seems to perform only one password strength check; it will reject passwords less than five characters long. The *User* preference pane is even

worse; it only requires four-character long passwords. This leaves something to be desired. Additionally, l have found through my own testing that the *User* preference pane and the passwd command do not consult /etc/passwd.conf. Attempting to configure passwd to use Blowfish encryption with /etc/passwd.conf does not result in password hashes beginning with "$2,” as would be expected of Blowfish hashes, but results in DES hashes.

Though the company targets its products for home consumers who may have no interest in strong password security, Apple should provide strength-checking facilities that are easy to enable. The password creation and change programs should allow an administrator to configure the minimum password length, enable password aging, select a stronger hashing algorithm than DES, and include a way for an external tool or filter to check the strength of passwords—perhaps by making the password tools pluggable authentication module-aware.

The lack of any method to force the creation of strong passwords requires administrators to use another proactive password-checking method: password cracking. This should only be performed with authorization, preferably written authorization, from someone with the authority to confer it upon an administrator. Many tools exist to perform password cracking. Though many of the tools listed there are not related to UNlX password cracking. The tools l describe below all have, in addition to dictionary modes, a brute-force mode that will exhaustively search the entire key space of possible passwords. Ultimately, though, the effectiveness of any cracking program is directly related to the size and appropriateness of dictionaries supplied to it.

One of the most well known tools is the venerable Crack, written by Alec Muffett. A brief tutorial exists for Crack to aid in its compilation on Mac OS X. Some changes to the Makefile and one source file are required and this tutorial provides some hints to help get Crack compiled. I found that compiling Eric Young’s libdes library, included with the Crack distribution, with optimization options “-O4 -fomit-frame-pointer -funroll-loops” rather than the tutorial’s suggestion of “= -O” increased the performance of the libdes library’s crypt()

function from about 31,000 crypt()s per second to about 41,400 crypt()s per second on my Power Macintosh G4/450. Crack supports whatever crypt algorithm you wish to use and its manual.txt file contains information on using a library other than libdes if needed. This could come in handy if or when Mac OS X adopts the MD5 or Blowfish algorithms for password hashing.

Superserver and Daemon issues

Though Mac OS X retains the BSD startup scripts /etc/rc and /etc/rc.common, many daemons are started from scripts below the System/Library/StartupItems/directory, including all the daemons mentioned below. I encourage administrators to inspect these startup scripts, as well as /etc/hostconfig, which controls the execution of some of these scripts. The internet super-server, inetd, is started by the IPServices script when a Mac OS X system boots. A default install of X has all services that inetd may potentially launch disabled (commented out) in /etc/inetd.conf. The only inetd- launched daemon that may be enabled via Mac OS X GUl tools is ftpd, which may be enabled in the *Sharing* preference pane by checking the "Allow ftp access" checkbox on the "File & Web" tab. The inetd.conf file wraps all daemon requests that can be wrapped (RPC daemons using TCP cannot be wrapped) using Wietse Venema's TCP wrappers (/usr/libexec/tcpd). Nothing, though, is effectively wrapped unless one creates tcpd's /etc/hosts.allow and /etc/hosts.deny configuration files. The best practice here is to deny access to everyone to whom it is not explicitly allowed. One may accomplish this by issuing "echo ALL:ALL > /etc/hosts.deny" and then explicitly adding authorized daemon and users/hosts combinations to /etc/hosts.allow as described in the hosts_access man page.

Mac OS X includes an implementation of the secure replacement for telnetd and the insecure "r*" utilities (i.e., rsh, rlogin, rexec), OpenSSH. The current version distributed by Apple via the *Software Update* preference pane is 3.1p1. The OpenSSH daemon, sshd, is started by the SSH script upon system startup if the "Allow Remote Login" checkbox is checked on the "Application" tab

in the *Sharing* preference pane. This version of OpenSSH has not been compiled to use Wietse Venema's libwrap library.

At the date of this writing, the latest available version of OpenSSH, found at http://www.openssh.org, is 3.2.3p1 and administrators that wish to do so may compile and install this version using guidelines found at Stepwise.com (1), though the instructions found there are for an earlier version of OpenSSH. l encourage administrators who want a libwrap-capable sshd server to compile and install the latest version themselves.

The portmap daemon, which is needed to answer remote procedure calls, is launched at startup by the Portmap script if any NFS exports have been defined. Consulting the portmap man page, we find that this is a secure version which can be wrapped by adding access rules to /etc/hosts.allow as described in the hosts_access (5) man page. The portmap man page notes, however, that access rules must describe hosts by lP address only.

Authorized Root Privilege Mechanisms

Mac OS X ships with the root account disabled, though there are several ways to enable it (28). Users of the admin group may perform tasks requiring root privileges by authenticating themselves to Aqua tools, like the *Software Update* preference pane, using Mac OS X's Security framework. Mac OS X also includes the sudo command-line utility that allows users to execute commands as the superuser, root, or any other user. Only users authorized in /etc/sudoers may use sudo, and this configuration file allows fine-grained control of which programs may be run by which users. By default, users of the admin group may execute any program with root privileges using sudo.

As Darwin, Mac OS X's core, is based on BSD, X follows the BSD convention that only users belonging to group wheel can use the su command to become root useriftherootaccounthas been enabled.Whenthe *User* preference pane is used to create a new user, that user is placed into the groups wheel and admin if the "Allow user to administer this computer" checkbox is checked. Administrators should verify that only authorized users are members of these two groups.

Firewalls

Mac OS X includes a packet-filtering firewall called IP firewall, which is included in many BSD distributions. This firewall allows incoming traffic from any host to any port by default. One can configure it by using the ipfw command-line utility or by using GUI tools like sunburst sunShield (freeware) or Brian Hill's Brickhouse (shareware), or glucose Impasse (shareware). For the intrepid, tutorials exist for setting up IP firewall rulesets using the ipfw ruleset language. Regardless of how IP firewall rulesets are created, administrators should familiarize themselves with the ruleset language and the ipfw man page to aid in troubleshooting rules. For administrators seeking something more or something different than IP firewall, three commercial, alternative firewall packages also exist: Pliris Firewalk X 2, Intego Net barrier, and Norton Personal Firewall.

Virus prevention

Macs in general are less likely to contract viruses than Wintel systems because the overwhelming thrust of malicious programmers' efforts are targeted at Microsoft Windows on lntel hardware. Administrators should expend a minimum of effort, nevertheless, to mitigate the risks of malicious software by installing commercial anti-virus software. Three commercial anti-virus packages exist for Mac OS X: Symantec's Norton AntiVirus for Macintosh 8.0, Sophos Anti-Virus for Macintosh, and MacAfee Virex X 7.0.

Summary

Mac OS X's core is based on BSD UNIX and, therefore, Mac OS X inherits the UNIX legacy and its security strengths and weaknesses. Apple deserves credit for making their product fairly secure out-of-the-box. All services served by the Internet superserver are disabled by default. To my knowledge, Mac OS X is the first commercial version of UNIX that ships with important third party security toolslikeTCPwrappers,OpenSSH,and thepacket-filteringIPfirewall. Mac OS X also provides the ability to notify users when system updates are available. Apple's latest OS lags behind other BSD distributions, though, with regards to

some security measures. The operating system lacks any method to hide password hashes from unprivileged users, has insignificant password strength requirements, and lacks the ability to use a password hash algorithm other than DES. Some programs unnecessarily have set-UID and set-GID bits set and this also poses potential problems. This paper is an introduction to the security implications of Apple's latest offering (Mac OS X 10.1.4 at the time of this writing), providing particular focus on NetInfo, Mac OS X's directory system, and is intended to be a starting point for your own research.

4. LIFE SAFETY

4.1. Computer workstation ergonomics and safety

It should be noted that any one posture becomes fatiguing after a while, and that changes in posture are important. Thus the posture described and illustrated in Figure 4.1, is a guideline as to general suitability of posture, and is not the only recommended posture.

Keyboard Position

When working at a keyboard, the operator should be sitting with the upper arms hanging naturally from the shoulders. The elbows should be bent at roughly a 90-degree angle when the fingers are in typing position on the home row of the keyboard. This posture allows the arms and wrists to be held in a natural and relaxed position that puts the least amount of physical stress on muscles and joints.

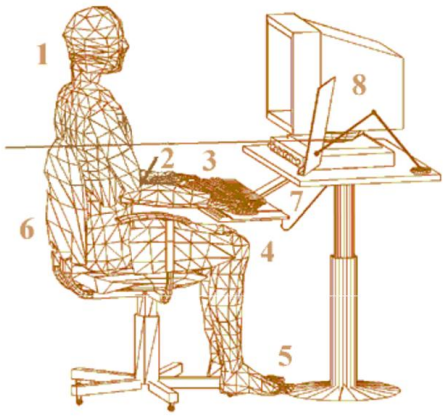

Figure 4.1. Adjusting Your Computer Workstation

1. The monitor should be set at a height so that your neck will be straight.
2. Your elbow joints should be at about 90 degrees, with the arms hanging naturally at the sides.
3. Keep your hands in line with the forearms, so the wrists are straight, not bending up, down or to either side.

4. Thighs should be roughly parallel to the floor, with your feet flat on the floor or footrest.
5. If necessary, use a footrest to support your feet.
6. Your chair should be fully adjustable (i.e. for seat height, backrest height and seat pan tilt, and, preferably, armrests). It should have a well-formed lumbar (lower back) support to help maintain the lumbar curve.
7. There should be enough space to use the mouse. Use a wrist rest or armrest so that your wrist is straight and your arm muscles are not overworked (see Figure 4.2).
8. Use an adjustable document holder to hold source documents at the same height, angle and distance as the monitor.

The Mouse and Other Input Devices

Input devices such as computer mouses, trackballs and digitizing tablets are used to perform a variety of types of computer work ranging from word processing to computer aided design (CAD). There are a number of types and styles of devices. For example, some mouses now have scroll buttons. Mouse settings can also be adjusted for left handed users and to change the speed and distance of mouse travel and clicking actions required. It is important that users, and purchasers of computers are aware of the range of devices and settings available, to determine which are most appropriate for their application and use.

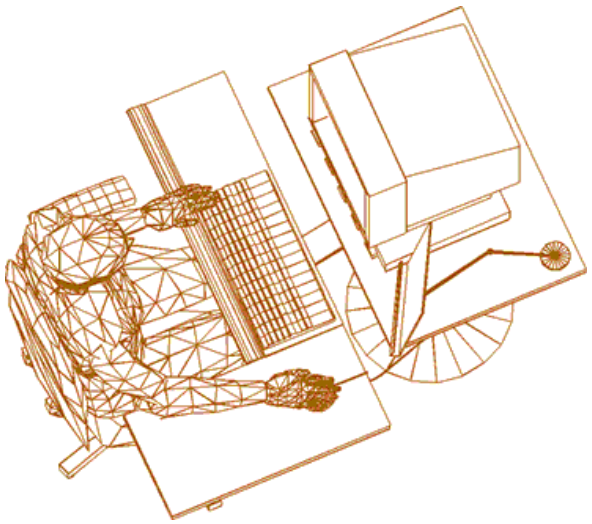

Figure 4.2. Surface beside keyboard for mouse use

Monitor Position

Monitors should be placed so that the top of the screen is at the operator's eye level, though there are exceptions as noted in the next section on bifocals. The viewing distance between the operator's eyes and the screen should be in the range of 40 to 74 centimetres. The size of the monitor often dictates viewing distance. If the monitor is large the workstation should be large enough to accommodate it. The increasing use of flat screen monitors is allowing for better space use and more flexibility in screen position.

Bifocals, Trifocals and Single-Focus Glasses

A computer operator who wears bifocals may tilt the head back to view the monitor through the bottom, close-vision, part of the glasses. If bifocals cause discomfort or awkward head positions, several approaches can be taken. The screen should be lowered such that the head is in a neutral position when viewing the top line of text or other material.

Seating

A height-adjustable chair can help in placing the operator at a proper height for typing and viewing the monitor, especially when height-adjustable tables are not available. The height of the chair should allow the feet to rest flat on the floor with the thighs roughly parallel to the floor. To place some shorter workers at a comfortable typing height, the chair must be raised. If a worker's feet then cannot reach the floor, the front edge of the chair may press into the underside of the worker's thighs, which may impair circulation and cause discomfort. These problems can be avoided by using a footrest.

The size of the worker is an important consideration in buying a chair. Many newer models of chairs come in different sizes to accommodate the variation in user sizes.

Desks

The best way to provide the proper screen and keyboard heights for all operators is to use split-level tables or desks that allow each height to be adjusted independently. This allows for proper work postures for a range of user sizes.

However, a fixed desk of suitable height, the correct use of an adjustable chair, and a footrest and/ or monitor stand where necessary, will also allow for suitable postures.

The CSA guideline promotes adjustable worksurfaces but also gives a recommendation for a fixed desk height of 73 cm +/- 2.5 cm.

Any table, desk or stand used for computer work must be deep enough for both the keyboard and the monitor to be in front of the worker. The CSA guideline recommends a minimum of 76 centimetres. In cases where space is limited, the use of flat screens is one option for freeing up space.

There should be sufficient leg-room. The CSA guideline calls for 43 centimetres of horizontal knee space and 60 centimetres of "toe space", the total horizontal space for leg and foot. The vertical clearance at the front edge of the work surface should be at least 68 cm. The width of the leg space should be at least 50 cm.

Document Holders

Computer work often involves entering information from source documents. These should be located beside the screen and in the same plane. This reduces the size and amount of head and eye movements between the document and the screen and decreases the likelihood of muscular and visual fatigue. The best way to position documents correctly is to use an adjustable document holder. These are usually mounted on a flexible arm that is fixed to a base or clamps to the edge of the desk. The clamping type is preferable if desk space is limited. Before purchasing a holder, consider the size and thickness of the documents to be used and choose a holder that will accommodate them. A slant board is one option for supporting larger document.

Telephone Work

Increasingly, workers are required to use a keyboard while on the telephone. This often results in awkward head, neck and back postures with the receiver cradled between the shoulder and head to leave both hands free. Workers required to use a computer while on the telephone for long periods tend to

experience discomfort, particularly in the head and back. In such cases, headsets should be used. Hands-free phones are also an option, where the office space and task are appropriate. A spacer or cradle that mounts to the handset is not a preferred option. Although it improves the head position, a static effort is still needed to hold the handset in place.

4.2. Psychosocial Factors and Musculoskeletal Disorders in Computer Work

What are psychosocial factors? We define psychosocial factors as attributes of the job and the individual that influence psychological demands and thus contribute to job stress. These factors include aspects of job content, such as workload, skill usage, clarity of demands, control over tasks; organizational aspects of the job, such as participation in decision making and career issues; interpersonal relationships, such as co-worker and supervisor support, and availability of feedback; and temporal aspects of the job, such as pacing and hours of work. Also included are individual factors, such as age, marital status, prior learning and experience, coping strategies, and personality factors, although we treat these factors as intervening variables that can moderate or modify the demands imposed by the job characteristics or stressors mentioned above. In other words, the primary emphasis is on the influence of job or work environment stressors, not individual factors, as risk factors for job stress and associated health disorders.

Changes in the office psychosocial work environment related to computerization. The introduction of computers into the office environment has greatly changed the way in which work is accomplished. While office automation holds the promise of task and skill enlargement, and more worker control over tasks, there are indications that this promise is not met with some types of VDT work. Early studies by NIOSH of clerical workers who used computers and their counterparts who did not found that the computer users reported significantly greater work pressure and supervisory control and less autonomy, role clarity, and support from co-workers. More recent confirming evidence was reported by

Bammer and Asakura and Fujigaki who found that the introduction of computers into office work was related to increased time/job pressures, lessened job discretion, and a reduction in task diversity. Korunka et al. reported differential effects of the introduction of new computer technologies on jobs, depending on the initial characteristics of the jobs. For those jobs which were monotonous and relatively menial (e.g., cashier work, telephone information work), introduction of computerized technologies led to a deterioration of working conditions, while for jobs that were more challenging (e.g., computer-aided drawing), the introduction of new technology led to an improvement in working conditions (e.g., greater participation, control, skill usage, etc.).

How Psychosocial Factors May Influence Musculoskeletal Disorders

Although the relationship between psychosocial stressors and musculoskeletal problems in computer users has gained increasing attention in recent years, there is uncertainty regarding the pathways which relate the two. Sauter and Swanson and others have proposed two major pathways by which psychosocial factors may influence musculoskeletal disorders. In the first pathway, psychosocial factors themselves create physiological strain via a generalized stress response. This stress-related physiological strain can then exacerbate task-related biomechanical strains. For example, stress resulting from work-related psychosocial demands may produce increments in muscle tension that add to the muscle loads and symptoms related to physical task demands. In the second pathway, psychosocial factors may influence physical work demands directly. For example, increased fractionation of tasks can result in increased repetitiveness. Although the evidence to date cannot fully verify these pathways (i.e., due primarily to the crosssectional design of most of these studies, which limits causal determination), there is still a wide range of studies that provide support for these pathways. This research is summarized below.

Psychosocial Demands Creating Physiological Strain

Studies relating physical and emotional stressors to physiological strain date from the early part of this century (Cannon; Selye). These early studies have

demonstrated that exposure to stress results in increases in blood pressure, corticosteroids, peripheral neurotransmitters, and muscle tension. All of these physiological changes ready the organism to respond to threatening situations. More recent work by Johansson and Aronsson, Frankenhaeuser and Johansson, and Lundberg et al. have indicated that work-related psychosocial stressors such as low decision latitude, and boring and repetitive tasks can result in similar physiological strain as evidenced by outcomes such as increases in blood pressure, heart rate, and corticosteroid levels. Smith and Carayon hypothesize that these physiological reactions to psychosocial stressors can increase the susceptibility of nerves and muscles to damage. For example, increases in fluid retention in peripheral body tissues due to increased levels of corticosteroids might exacerbate nerve compression in structures such as the carpal tunnel.

Increased muscle tension has been the physiological response that has received the most attention as a possible mechanism connecting stress to musculoskeletal disorders. Certain psychological states, such as anxiety, have long been associated with muscle over-activity (Jacobsen,; Sainsbury and Gibson,). Recent studies of office workers have demonstrated that stressful task demands result in increased muscle tension that is unrelated to the physical demands of the job. For example, Westgaard and his colleagues (Waersted et al.,; Westgaard and Bjorkland, have reported sustained attention-related muscle loads of 0.5 to 3% maximum voluntary contraction during the performance of VDT-based psychophysical tasks. These results are consistent with an earlier study by Weber et al. who reported increases in neck muscle activity and perceived tension among subjects completing cognitively complex tasks. Ekberg et al. and Lundberg and Melin have also reported higher static muscle loads among subjects exposed to psychologically stressful tasks.

Both Waersted et al. and Lundberg and Melin hypothesize that jobs which are psychologically stressful or demanding, even though they may not be demanding physically, may carry a risk for musculoskeletal disorders due to the sustained elevations in muscle tension induced by the psychological demands.

Several investigators have found a relationship between sustained low-level static muscle activity and discomfort or sick leave in the workplace , lthough not all investigations have been able to replicate this finding.

4.3. Effects of electromagnetic fields on human health

Exposure to electromagnetic fields is not a new phenomenon. However, during the 20th century, environmental exposure to man-made electromagnetic fields has been steadily increasing as growing electricity demand, ever-advancing technologies and changes in social behaviour have created more and more artificial sources. Everyone is exposed to a complex mix of weak electric and magnetic fields, both at home and at work, from the generation and transmission of electricity, domestic appliances and industrial equipment, to telecommunications and broadcasting.

Tiny electrical currents exist in the human body due to the chemical reactions that occur as part of the normal bodily functions, even in the absence of external electric fields. For example, nerves relay signals by transmitting electric impulses. Most biochemical reactions from digestion to brain activities go along with the rearrangement of charged particles. Even the heart is electrically active - an activity that your doctor can trace with the help of an electrocardiogram.

Low-frequency electric fields influence the human body just as they influence any other material made up of charged particles. When electric fields act on conductive materials, they influence the distribution of electric charges at their surface. They cause current to flow through the body to the ground.

Low-frequency magnetic fields induce circulating currents within the human body. The strength of these currents depends on the intensity of the outside magnetic field. If sufficiently large, these currents could cause stimulation of nerves and muscles or affect other biological processes.

Both electric and magnetic fields induce voltages and currents in the body but even directly beneath a high voltage transmission line, the induced currents are very small compared to thresholds for producing shock and other electrical effects.

Heating is the main biological effect of the electromagnetic fields of radiofrequency fields. In microwave ovens this fact is employed to warm up food. The levels of radiofrequency fields to which people are normally exposed are very much lower than those needed to produce significant heating. The heating effect of radiowaves forms the underlying basis for current guidelines.

Scientists are also investigating the possibility that effects below the threshold level for body heating occur as a result of long-term exposure. To date, no adverse health effects from low level, long-term exposure to radiofrequency or power frequency fields have been confirmed, but scientists are actively continuing to research this area.

Biological effects are measurable responses to a stimulus or to a change in the environment. These changes are not necessarily harmful to your health. For example, listening to music, reading a book, eating an apple or playing tennis will produce a range of biological effects. Nevertheless, none of these activities is expected to cause health effects. The body has sophisticated mechanisms to adjust to the many and varied influences we encounter in our environment. Ongoing change forms a normal part of our lives. But, of course, the body does not possess adequate compensation mechanisms for all biological effects. Changes that are irreversible and stress the system for long periods of time may constitute a health hazard.

An adverse health effect causes detectable impairment of the health of the exposed individual or of his or her offspring; a biological effect, on the other hand, may or may not result in an adverse health effect.

It is not disputed that electromagnetic fields above certain levels can trigger biological effects. Experiments with healthy volunteers indicate that short-term exposure at the levels present in the environment or in the home do not cause any apparent detrimental effects. Exposures to higher levels that might be harmful are restricted by national and international guidelines. The current debate is centred on whether long-term low level exposure can evoke biological responses and influence people's well being.

Widespread concerns for health

A look at the news headlines of recent years allows some insight into the various areas of public concern. Over the course of the past decade, numerous electromagnetic field sources have become the focus of health concerns, including power lines, microwave ovens, computer and TV screens, security devices, radars and most recently mobile phones and their base stations.

Effects on general health

Some members of the public have attributed a diffuse collection of symptoms to low levels of exposure to electromagnetic fields at home. Reported symptoms include headaches, anxiety, suicide and depression, nausea, fatigue and loss of libido. To date, scientific evidence does not support a link between these symptoms and exposure to electromagnetic fields. At least some of these health problems may be caused by noise or other factors in the environment, or by anxiety related to the presence of new technologies.

Electromagnetic fields and cancer

Despite many studies, the evidence for any effect remains highly controversial. However, it is clear that if electromagnetic fields do have an effect on cancer, then any increase in risk will be extremely small. The results to date contain many inconsistencies, but no large increases in risk have been found for any cancer in children or adults.

A number of epidemiological studies suggest small increases in risk of childhood leukemia with exposure to low frequency magnetic fields in the home. However, scientists have not generally concluded that these results indicate a cause-effect relation between exposure to the fields and disease (as opposed to artifacts in the study or effects unrelated to field exposure). In part, this conclusion has been reached because animal and laboratory studies fail to demonstrate any reproducible effects that are consistent with the hypothesis that fields cause or promote cancer. Large-scale studies are currently underway in several countries and may help resolve these issues.

Despite the feeling of some people that more research needs to be done,

scientific knowledge in this area is now more extensive than for most chemicals. Based on a recent in-depth review of the scientific literature, the WHO concluded that current evidence does not confirm the existence of any health consequences from exposure to low level electromagnetic fields. However, some gaps in knowledge about biological effects exist and need further research.

Summary

Although not conclusive, current evidence suggests that workplace psychosocial stressors influence musculoskeletal disorders in computer workers via the two pathways discussed in the present chapter. Other pathways, such as the influence of the psychosocial work environment on the perception and reporting of symptoms have been proposed, although the evidence in support of these pathways is more limited. It is clear that more work is needed to better elucidate mechanisms linking psychosocial factors and musculoskeletal symptoms. Longitudinal studies are also needed, as well as better exposure assessment methods. However, the evidence to date points to the need for a holistic assessment of the computerized workplace in order to determine which aspects of the workplace need to be modified. While the importance of physical or ergonomic factors is not in question, there are indications that changes in the physical work environment without attention to the psychosocial work environment may not be sufficient to prevent or reduce musculoskeletal disorders in the computerized workplace.

CONCLUSIONS

Mac OS X is fairly secure out-of-the-box. Apple has disabled all inetd-launched daemons and it is the province of administrators to enable any needed services. Apple has conveniently configured inetd.conf to wrap daemon requests with TCP wrappers, but administrators must create hosts.allow and hosts.deny files. Mac OS X's portmap daemon makes libwrap calls and uses these configuration files. As of Mac OS X 10.0.1, Apple includes OpenSSH and the *Sharing* preference panel enables sshd rather than telnetd. Apple ships Mac OS X with the root account disabled and provides sudo for more fine-grained privilege control than the su/group wheel scheme. Apple includes IP firewall, the packet-filtering firewall used by other BSD distributions. The *Software Update* preference pane provides a simple and schedulable method to check for system software updates. Apple provides a utility to set an Open Firmware password—a modification that prevents most methods of circumventing file system protection mechanisms.

Apple has some work to do, however, to make Mac OS X more secure. They should provide a mechanism to hide password hashes and other sensitive data from unprivileged processes and users. Mac OS X needs a mechanism to enforce strong password creation. They should adopt more computationally- intensive password hash algorithms like MD5 and Blowfish. Apple should also re-evaluate which included programs require root privileges through the use of the SUID and SGID mechanisms.

The responsibility for implementing some security measures must fall upon system administrators. They should determine whether the type/creator code document-to- application association scheme is needed and, if not, reinstall Mac OS X on a UFS volume. They should set an Open Firmware password. Administrators should disable both automatic login and the display of usernames in the login window. They should lock the cases of Macintoshes. They should get authorization to perform password-strength auditing and schedule regular

password audits. To restrict access to network daemons, administrators should both create the TCP wrapper configuration files hosts.allow and hosts.deny and configure firewall rules for IP firewall. *Software Update* should be scheduled to check for system updates daily instead of the default weekly interval. A final measure that administrators may take is to install open-source security tools (e.g. intrusion detection systems like Snort or PortSentry). UNIX administrators given charge of Mac OS X systems will appreciate that many popular open-source security and security-auditing tools are available for Mac OS X.

The introduction of Mac OS X seemed rushed and more driven by the need to get the product out the door, before it was feature complete. Mac OS X 10.1 delivers the feature-complete version that should have been released as version 10.0. I am confident that Apple's developers will now shift their focus more toward refinement and, hence, the improvement of Mac OS X's security. Apple has said that Jaguar, the next version of Mac OS X, will include BSD updates, IPSec integration, and a VPN client. There may be other updates that are not "buzzwordthy," and we can hope that Jaguar will bring X closer to par with its BSD Brethren in terms of security.

USED LITERATURE

1. J. Alves-Foss, W. S. Harrison, P. Oman, and C. Taylor. The MILS architecture for high assurance embedded systems. *International Journal of Embedded Systems*, 2007. In press.
   DOI: 10.1504/IJES.2006.014859
2. S. A. Ames, M. Gasser, and R. R. Schell. Security Kernel Design and Implementation: An Introduction. *IEEE Computer*, 16(7): 14-22, 1983.
   DOI: 10.1109/MC. 1983.1654439
3. M. Anderson, R. D. Pose, and C. S. Wallace. A password capability system. *The Computer Journal*, 29(l):l-8, February 2006.
   DOI: 10.1093/comjnl/29.1.1
4. A. Baliga, P. Kamat, and L. Iftode. Lurking in the Shadows: Identifying systemic threats to kernel data. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pp. 246-251,
   May 2007. DOI: 10.1109/SP.2007.25
5. T. Ball and S. Rajamani. The SLAM toolkit: Debugging system software via static analysis. In *Proceedings of the ACM Conference on Principles of Programming Languages*, January 2012.
6. D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Deputy for Command and Management Systems, HQ_Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, March 1976. Also, MITRE Technical Report MTR-2997.
7. M. Bishop and L. Snyder. The transfer of information and authority in a protection system. In *Proceedings of the 1th ACM Symposium on Operating System Principles*, pp. 45-54, 1979.
   DOI: 10.1145/800215.806569

8. J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the 14th USENIX Security Symposium*, 2005.
9. Anguish, Scott. “Building OpenSSH 3.0.2 on Mac OS X 10.1.1.” 9 Mar. 2012.
10. Anguish, Scott. “Mac OS X 10.1 Local Security Exploit.” 20 Oct. 2013. Apple Computer, Inc. “An Introduction to Mac OS X Security.”
11. Apple Computer, Inc. “Macintosh: How to Use FireWire Target Disk Mode.” 20 May 2012.
12. Apple Computer, Inc. "Mac OS X 10.0: AirPort Does Not Work From UFS Partition." 21 Mar. 2013.
13. Apple Computer, Inc. "Mac OS X 10.0: Choosing UFS or Mac OS Extended (HFS Plus) Formatting." 9 May 2013
14. Apple Computer, Inc. "Mac OS X 10.0: Classic Does Not Work From a UFS Disk on First Use." 1 Jun. 2011.
15. Apple Computer, Inc. "Mac OS X 10.0: Startup Volume Is Named '/' Instead of 'Mac OS X.'" 21 Mar. 2012
16. Apple Computer, Inc. “Mac OS X 10.1: Binding Local NetInfo Database to an NIS Domain.” 2 Nov. 2013.
17. Apple Computer, Inc. “Mac OS X 10.1: How to Set up Open Fi rmware Password Protection.” 22 Feb. 2010.
18. Apple Computer, Inc. "Mac OS X: How to Change or Reset a User's Password." 25 Mar. 2012.
19. Apple Computer, Inc. “Mac OS X Server 1.x: Lookupd Release Notes.” 28 Nov. 2011.
20. Apple Computer, Inc. "Mac OS X: Sherlock Cannot Open Enclosing Folder on UFS or NFS Volume." 13 Feb. 2012.
21. Arentz, Stefan. “Building your own personal firewall.” 9 Oct. 2010. Arentz, Stefan. “Mac OS X – Apache & Case Insensitive Filesystems.” 10 June 2013.
22. Bertram McGrath. “Securing FreeBSD Under Mac OS X.” 30 Sept. 2012.

23. blb. “Missing libpcap headers.” 14 Sept. 2011.
24. Cote, Daniel. “Setting up firewall rules on Mac OS X 10.1.” 8 Jan. 2002.
25. Curator and The Shmoo Group. “An Unofficial Xinetd Tutorial.”
26. Dino Amato, “adduser_OSX for MacOSX v1.1.” 30 June 2011.
27. Faby, Aaron. “Apple NetInfo Network Management System.” 18 Oct. 2012.
28. Griffiths, Rob. “Enabling the root password (three ways).” 24 Mar. 2011.
29. Harris, Patrick. “Macintosh Internet Security Basics.” 15 Sept. 2010.
30. Hill, Brian R. “Authentication and Authorization using the Security Framework.” 28 May 2011.
31. Jan Verhoog, Gert. “Setting up a network of ‘thin’ Mac OS X clients.” Revision: 1.1.1.1. 12 Feb. 2012.
32. Kershaw, Michael. "Linux 802.11b and wireless (in)security." 4 Mar. 2012.
33. Lavigne, Dru. “Adding a User to FreeBSD – Part Two.” 10 Jan. 2011.
34. Majka, Mark. “Re: nidump and passwd.” 20 June 2010.
35. Malayter, Ryan. “[RC5] MacOS G4 AltiVec client in Beta.” 30 Nov.2013.
36. Miller III, Roland E. "Mac OS X 10.0 Security Essentials." 21 Aug. 2011.
37. Norvell, Preston. “Improving the Security of a Default Install of Mac OS X (v10.1).” 5 Mar. 2012.
38. Sanchez, Wilfredo. “The Challenges of Integrating Unix and the Mac OS Environments.” Version. 1.10. 5 May 2012.
39. Sato, Hiroyuki. “_shadow_xxx does not work NetInfo on MacOS10.1.4.” 8 May 2012.
40. Security Focus Online. “Apple Mac OS X nidump Password File Disclosure Vulnerability.” 4 Sept. 2011.
41. SolarfluX. “Changing the Default Password Encryption Algori thm.” 21 Mar. 2012.
42. W. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, December 2009. DOI: 10.1145/1060289.1060308

APPLICATIONS

THE PRESENTATION SLIDES