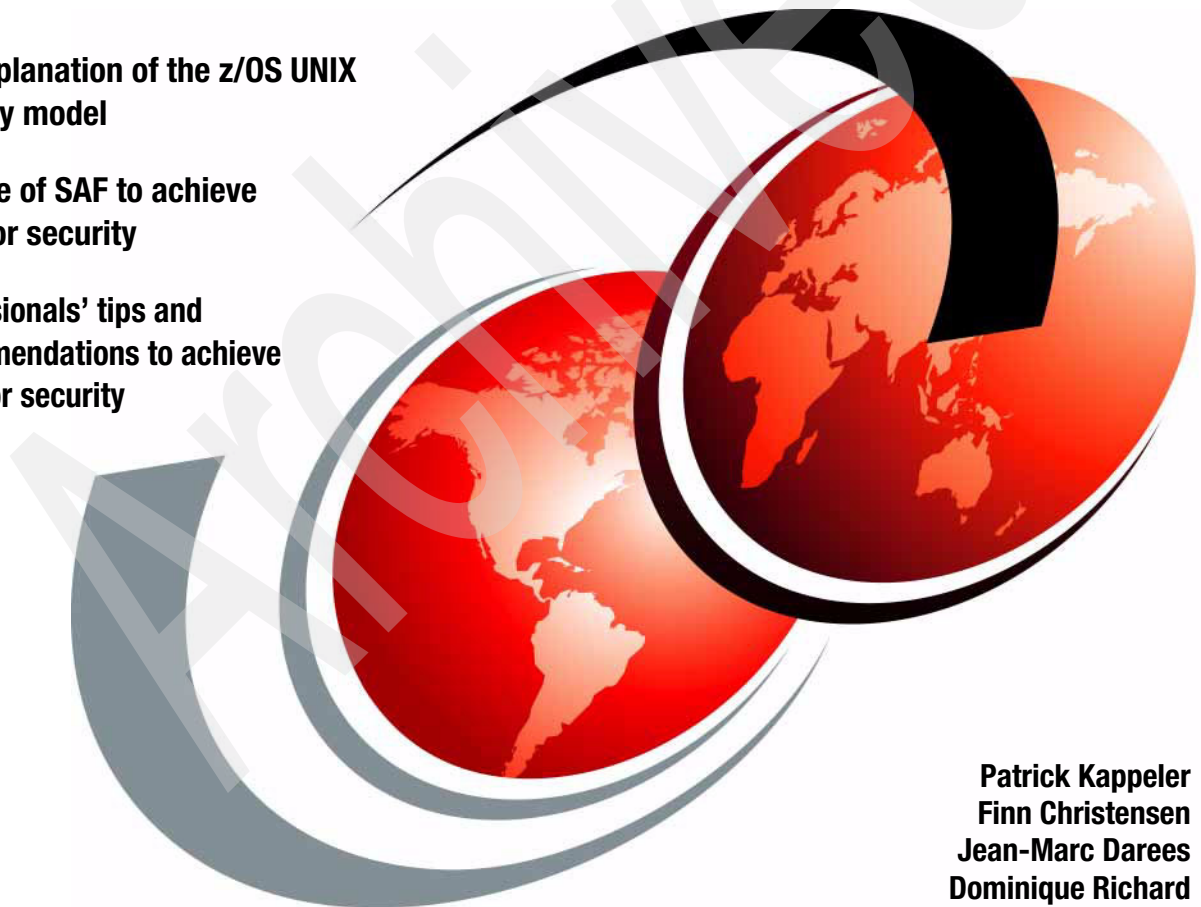


z/OS UNIX Security Fundamentals

The explanation of the z/OS UNIX security model

The use of SAF to achieve superior security

Professionals' tips and recommendations to achieve superior security



Patrick Kappeler
Finn Christensen
Jean-Marc Darees
Dominique Richard



International Technical Support Organization

z/OS UNIX Security Fundamentals

February 2007

Archived

Note: Before using this information and the product it supports, read the information in “Notices” on page vii.

First Edition (February 2007)

This edition applies to the z/OS UNIX System Services base component in Version 1, Release 7, of z/OS (product number 5694-A01)

© Copyright International Business Machines Corporation 2007. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	vii
Trademarks	viii
Preface	ix
The team that wrote this IBM Redpaper	ix
Become a published author	x
Comments welcome	xi
Chapter 1. Overview of the UNIX operating system model	1
1.1 What is UNIX?	2
1.1.1 The POSIX standards	2
1.2 The UNIX model of operating system	3
1.2.1 The UNIX kernel	4
1.2.2 The UNIX processes	4
1.2.3 Signals	5
1.2.4 Virtual memory and memory protection	5
1.2.5 Shell	5
1.2.6 The UNIX utilities	6
1.2.7 The UNIX file system	6
1.2.8 The /etc directory	7
1.2.9 Daemons	8
1.3 The UNIX security model	8
1.3.1 Accessing UNIX	8
1.3.2 UNIX users and groups	8
1.3.3 File and directory permissions	10
1.4 The z/OS UNIX System Services history	11
Chapter 2. Overview of z/OS UNIX implementation	13
2.1 z/OS UNIX System Services fundamentals	14
2.1.1 Dubbing	15
2.1.2 z/OS UNIX services	16
2.1.3 z/OS UNIX and z/OS features	17
2.1.4 Resource Measurement Facility	20
2.1.5 z/OS UNIX configuration parameters	21
2.1.6 z/OS UNIX kernel	21
2.1.7 z/OS UNIX file system	21
2.2 Securing the z/OS UNIX environment	26
2.2.1 z/OS UNIX address spaces	26
2.2.2 HFS and zFS data sets	29

2.2.3	Protecting the BPXPRMxx member	30
2.2.4	Protecting z/OS UNIX related operator commands	30
2.3	Applications security: UNIX security and z/OS UNIX security	30
2.4	RACF AIM	30
Chapter 3. z/OS UNIX users and groups identity management		31
3.1	User identification and authentication in z/OS UNIX	32
3.1.1	User identity implementation	32
3.1.2	User authentication	33
3.2	The UID and GID in z/OS UNIX	34
3.2.1	The OMVS segment in the RACF USER profile	34
3.2.2	RACF group and z/OS UNIX	37
3.3	Default UID and GID	38
3.4	Shared UID and GID	40
3.4.1	Automatic prevention of UID sharing	40
3.4.2	Allowing assignment of shared UIDs or GIDs	41
3.5	Automatic UID and GID assignment	42
3.5.1	Specifying automatic assignment of UIDs and GIDs	42
3.5.2	Automatic UID and GID assignment in an RRSF configuration	43
Chapter 4. z/OS UNIX task identity management		45
4.1	Implementation of the UNIX process and threads concepts	46
4.1.1	The UNIX process	46
4.1.2	The UNIX thread	47
4.2	Identities associated with a z/OS UNIX process or thread	48
4.2.1	Real and effective UID and GID	49
4.2.2	The saved UID and saved GID	49
4.3	Functions that change the effective UID and GID	49
Chapter 5. The z/OS UNIX security model		51
5.1	The superuser concept and privileges	52
5.1.1	The concerns with the superuser concept	52
5.2	z/OS UNIX implementation of the superuser concept and privileges	53
5.2.1	Reminder on z/OS UNIX identity switching	53
5.2.2	Authentication of the switched-to user ID	56
5.2.3	The RACF BPX.DAEMON profile in the FACILITY class	57
5.3	Introducing the controlled environment	58
5.4	Using surrogate users with z/OS UNIX	61
5.5	BPX.SERVER	62
5.6	z/OS UNIX users privilege granularity	64
5.6.1	BPX.SUPERUSER	64
5.7	Individual limits in the USER profiles	65
5.7.1	The UNIXPRIV class of resources	65
5.8	Some recommendations	68

5.9 Other restrictions to superuser authority	69
5.10 The daemons in z/OS	69
5.11 Advanced topic: RACF enhanced program security	69
5.11.1 Overview of the principles of operation	69
5.11.2 Enhanced program security and z/OS UNIX	71
5.12 A word on IPC security	71
Chapter 6. z/OS UNIX files security	73
6.1 z/OS implementation of the Hierarchical File System	74
6.1.1 The z/OS UNIX file systems	74
6.1.2 Protection of the file system data sets	75
6.1.3 Mount security	76
6.2 UNIX files and directories security	77
6.2.1 The file security packet	77
6.3 File and directory access control permission bits	81
6.3.1 Default permission bits	84
6.3.2 The chmod command	85
6.3.3 Default owning UID and GID	86
6.4 File and directory access control: Access control list	86
6.5 File and directory access control: Security checks	89
6.5.1 Authorization checking algorithm without using an ACL	90
6.5.2 Authorization checking algorithm with ACL defined	92
6.6 The IRRHFSU utility	94
Chapter 7. Overview of multilevel security	97
7.1 The MLS security model	98
7.1.1 Applying MAC	98
7.1.2 Security labels	99
7.1.3 Domination and equivalence of security labels	100
7.1.4 Turning on MLS in RACF	101
7.1.5 When MLS is on	101
7.2 MLS and z/OS UNIX resources and users	102
Chapter 8. Considerations on z/OS UNIX program management	105
8.1 How to link-edit program into HFS files	106
8.2 Owner information for a z/OS UNIX file	106
8.3 Extended attributes of an HFS file	107
8.3.1 Program control bit	107
8.3.2 APF bit	108
8.3.3 The shared space bit	108
8.3.4 The library bit	109
8.4 The file mode section of the FSP	109
8.4.1 The non-permission bits	109
8.4.2 The permission bits	110

8.5 The sanction list	111
Chapter 9. Auditing z/OS UNIX	113
9.1 Overview of auditing options	114
9.2 File-based auditing options	115
9.3 Events always audited	116
9.3.1 RACF classes for auditing	116
9.4 Auditing for superuser authority and UNIXPRIV class privileges	118
9.5 Auditing reports	119
Appendix A. BPX. RACF profiles	121
Appendix B. C/C++ functions and UNIX System Services callable services	125
Related publications	127
IBM Redbooks	127
Other publications	127
Online resources	128
How to get IBM Redbooks	128
Help from IBM	128
Index	129

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

BookManager®

DB2®

DFS™

IBM®

Language Environment®

Lotus Notes®

Lotus®


MVS/ESA™

MVS™

Notes®

OS/390®

RACF®

Redbooks (logo) ™

Redbooks™

RMF™

System z™

Tivoli®

z/OS®

The following terms are trademarks of other companies:

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Preface

This IBM® Redpaper introduces the z/OS® UNIX® security model and implementation to IBM MVS™ knowledgeable and security-minded users. It does not address in detail all the wealth of specific security features available in z/OS UNIX, but rather the base principles of operation and the mechanisms implementation with setup recommendations.

We assume that the user already has a knowledge of the most commonly used IBM Resource Access Control Facility (RACF®) setups and commands. However, we do not provide detailed procedures and explanations about the use of these commands.

The team that wrote this IBM Redpaper

This IBM Redpaper was produced by a team of specialists from around the world working in the European Products and Solutions Center (Montpellier, France) on behalf of the International Technical Support Organization (ITSO) in Poughkeepsie (U.S.).

Patrick Kappeler led this IBM Redpaper project. For the past 35 years, he has been holding many international specialist and management positions in IBM, all dealing with mainframes Technical Support. He is now part of the European Products and Solutions Support Center, located in Montpellier (France), where his domain of expertise is the e-business Security on IBM System z™. He has authored many IBM Redbooks™ and still extensively writes and presents on this topic.

Finn Christensen is an IBM Senior I/T specialist. He has been an MVS Systems Programmer for nine years and with the IBM EMEA Cryptographic Competency Center in Copenhagen for another 15 years. He has made migration tools from TopSecret and ACF2 to RACF and has done many such migrations in many countries. He also does penetration tests as a fee service, and regularly presents on this subject at conferences. He has taken part in many residencies on mainframe-related security.

Jean-Marc Darees joined IBM in 1984 as an MVS System Engineer. Since this time, he has held several specialist and architect positions dealing with mainframe and other technologies supporting customer and internal projects. He joined the PSSC in Montpellier in 1997, where he now provides consulting

and presales technical support in the area of Siebel® CRM infrastructure for large customers.

Dominique Richard is an IT Specialist in IBM France. He joined IBM in 1982 and was a System Engineer supporting MVS customers in France. Since 2005, he is part of the European Products and Solutions Support Center, located in Montpellier (France), where he is involved in benchmarks. He is specialized in the area of host system security.

Thanks to the following people for their contributions to this project:

Chris Rayns
ITSO, Poughkeepsie Center

Bruce Wells
z/OS Security Server Development

Alain Roger
Pascal Tillard
Montpellier European Products and Solutions Support Center

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you'll develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our papers to be as helpful as possible. Send us your comments about this Redpaper or other Redbooks in one of the following ways:

- ▶ Use the online **Contact us** review redbook form found at:

ibm.com/redbooks

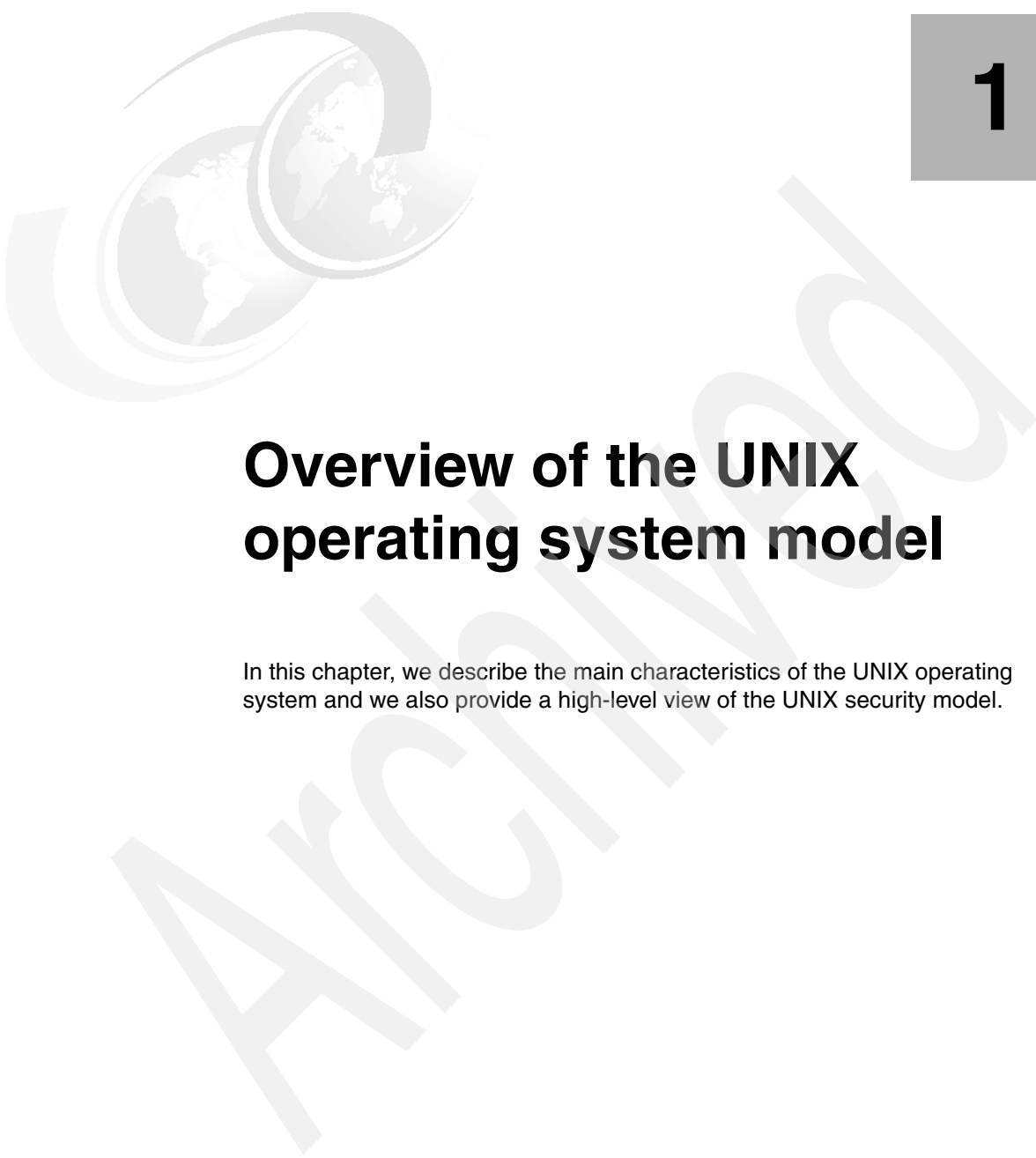
- ▶ Send your comments in an email to:

redbook@us.ibm.com

- ▶ Mail your comments to:

IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Archived



Overview of the UNIX operating system model

In this chapter, we describe the main characteristics of the UNIX operating system and we also provide a high-level view of the UNIX security model.

1.1 What is UNIX?

UNIX is an interactive, multi-user, multitasking operating system, designed in the 70s with the objective of providing a hardware platform independent operating system, thus allowing users to leverage their investments both in system programming, system management and application development skills as they were changing hardware platforms for scalability, quality of services, or for both these reasons.

Eventually UNIX became the prototype of what is now called an *Open System* because, at least in its mind-set, it met today's Institute of Electrical and Electronics Engineers (IEEE) definition of an Open System:

“An Open Systems environment is a comprehensive and consistent set of international information technology standards and functional profiles that specify interfaces, services, and supporting formats to accomplish Inter operability and portability of application, data, and people.”

The first version of UNIX was created in 1971. The term UNIX is not itself an acronym, but it was derived from the acronym of an earlier operating system called UNiplexed Information and Computing Service (UNICS).

UNIX has been extraordinarily well received in the data processing community; it happened to become the operating system of choice for small and medium systems, about a decade before personal computers came to be a commonly used device. It had the potential robustness that fitted most of the small and medium enterprises needs, thus leading to the constitution of a huge base of applications developed according to the UNIX model of operating system.

These days, UNIX is a registered trademark licensed exclusively through The Open Group. Operating systems may only use the UNIX trademark if they have been certified to do so by The Open Group, the condition for certification being to meet with the proposed implementation the thoroughly formalized system model behavior and to provide the strictly specified applications and user interfaces.

UNIX-compatible operating systems that are not certified by The Open Group are typically referred to as *UNIX-like*. For instance, Linux® is a UNIX-like operating system.

1.1.1 The POSIX standards

The IEEE definition of Open System and the lessons learned with the many different attempts to provide UNIX-like operating systems led to defining a family of standards to better define the characteristics of such a platform neutral operating system: the Portability System Interface (POSIX).

The POSIX standards cover a wide spectrum of operating system components ranging from C language and shell interfaces to system administration, and as such define an application programming interface (API), which could be supplied not only by UNIX systems but by any other operating systems, as it is the case for z/OS today, which implements the POSIX defined APIs. Note, however, that implementations of POSIX can be different in areas such as performance, availability, and recoverability, resulting in all POSIX-compliant systems not offering the same environmental characteristics although they all support basically the same interfaces.

The POSIX standards specify three different areas of compliance:

- ▶ POSIX.0: Relates to standards project and draft guide to the POSIX (p1003.0) Open System Environment.
- ▶ POSIX.1: Relates to System Application Program Interface (API) (p1003.1)
- ▶ POSIX.2: Relates to Shells and Utilities(p1003.2)

These POSIX standards have been incorporated since then into the XPG4 (X/Open Portability Guides) super-set of standards.

1.2 The UNIX model of operating system

In this section, we describe the main components of the UNIX operating system model.

As explained in the previous section, UNIX has been designed with portability in mind. This translated into:

- ▶ The operating system code being written in the C language (rather than a specific assembly language) so that it can easily be moved from one hardware platform to another.
- ▶ Moving an application (*or porting*) from one hardware platform to another is generally as simple as transferring the source code, then recompiling it.

UNIX also provides the following operating system features:

- ▶ Multitasking with virtual memory support and address spaces isolation.
- ▶ All users must be authenticated by a valid account and password to use the system. Files are owned by particular accounts. The owner can decide whether others have read or write access to the files he or she owns.
- ▶ Tooling and command language for system and application developments. One peculiar feature of UNIX is the huge amount of different user commands.

- ▶ A unified file system to represent data, programs, and any input or output ports used to transfer data as files, nested in directories, in a hierarchical tree structure.
- ▶ Support for distributed processing.

UNIX has three large functional blocks: The kernel, the shell, and the utilities. Technically, only the kernel and the shell form the operating system, while the utilities are intended to make the operating system more immediately useful to the user.

1.2.1 The UNIX kernel

The *kernel* is the core of the UNIX operating system. It consists of the collection of software modules that makes it possible for the operating system to provide services. The basic services provided by the kernel are:

- ▶ Creation and management of processes
- ▶ The file system
- ▶ The communications system
- ▶ The operating system start up processes

The kernel functions are of two broad types: Autonomous and responsive. Kernel functions, such as allocation of memory and CPU, are performed without being explicitly requested by user processes, therefore *autonomous*. Other functions of the kernel, such as resource allocation and process creation and management, are initiated by explicit requests from processes.

UNIX users are not required to have some kernel knowledge in order to use the system.

1.2.2 The UNIX processes

A *process* is the flow of execution of a set of program instructions and owns, as a system entity, the necessary resources. Some operating systems, such as z/OS, call the basic unit of execution a *job* or *task*. In UNIX, it is called a process. In the UNIX kernel, anything that is done, other than autonomous operations, is done by a process that issues system calls. Processes often spawn other *child* processes, using for instance the *fork()* system call, which usually run in parallel with their *parent* process. These are usually subtasks which, when they are finished, terminate themselves.

All UNIX processes have an *owner*. Typically, the human owner of a process is the owner of the account whose login process spawned the initial process parent of the process chain currently executing. The child process inherits the file access and execution privileges belonging to the parent.

Every process in UNIX is identified by a process ID (PID) number.

1.2.3 Signals

Signals are designed for processes to communicate with each other and with the kernel. The signalling capability is provided by the operating system and is used, for instance, to inform processes of unexpected external events such as a timeout or forced termination of a process. A signal consists of a prescribed message with a default action embedded in it. There are different types of signals in UNIX, and each type is identified with a number.

1.2.4 Virtual memory and memory protection

UNIX uses paging and swapping techniques similar to z/OS. Each UNIX address space has its own individual address translation table that thereby ensures isolation between address spaces.

1.2.5 Shell

The *shell* is the interactive environment that UNIX users are put in when they log in to the system. To perform work from the shell, the user enters commands in the shell screen.

The shell is a command interpreter, that is, it takes each command that is entered and passes it to the operating system kernel to be acted upon. The results of the system's operation is then displayed on the screen. Several command interpreters, that is several different shells, might be hosted by a UNIX system for the users to choose from. A user might decide to use the default shell or override the default to get access to another shell he or she prefers for some reasons. Some of the more common UNIX shells are:

- ▶ Bourne shell (the shell executable has usually a file extension "sh")
- ▶ C shell (csh)
- ▶ Korn shell (ksh)
- ▶ TC shell (tcsh)
- ▶ Bourne Again shell (bash)

Each shell also includes its own command programming language as Time Sharing Option (TSO) does with the CLIST or REXX languages. Command files, called *shell scripts*, can be built and invoked to automatically accomplish a series of tasks.

There is a graphical shell available for UNIX systems, called *X-Windows* or simply *X*. This graphical user interface (GUI) has all the features found on a personal computer. In fact, the version used most commonly on modern UNIX

systems (Common Desktop Environment (CDE)), is sponsored by The Open Group and is intended to give a common GUI look and feel to all open systems.

1.2.6 The UNIX utilities

UNIX includes many utility programs, often invoked as commands, to perform functions such as:

- ▶ Editing
- ▶ Maintaining files
- ▶ Printing
- ▶ Sorting
- ▶ Program debug
- ▶ Getting systems-related information

1.2.7 The UNIX file system

The UNIX file system hosts the collection of files accessed by the processes running in the system and is in charge of the logical representation of the data to the requesting entities. The file system has therefore both a logical and physical dimension.

The logical file system

The logical file system is in charge of the hierarchy of connected directories and files as they are shown to the users. The UNIX file system is logically arranged as a *tree*, actually inverted with the root, named “/”, at the top. All files are logically contained within the root directory. See the example shown in Figure 1-1 on page 7, where the shaded boxes represent directories, while the unshaded boxes represent files. A file or directory is located in the file system tree using a “path name”; */etc/profile* or */u/dirA/dirA1/Dominique* are path names.

Note that UNIX is a case-sensitive operating system, therefore a file called “ABC” is different from a file called “abc”.

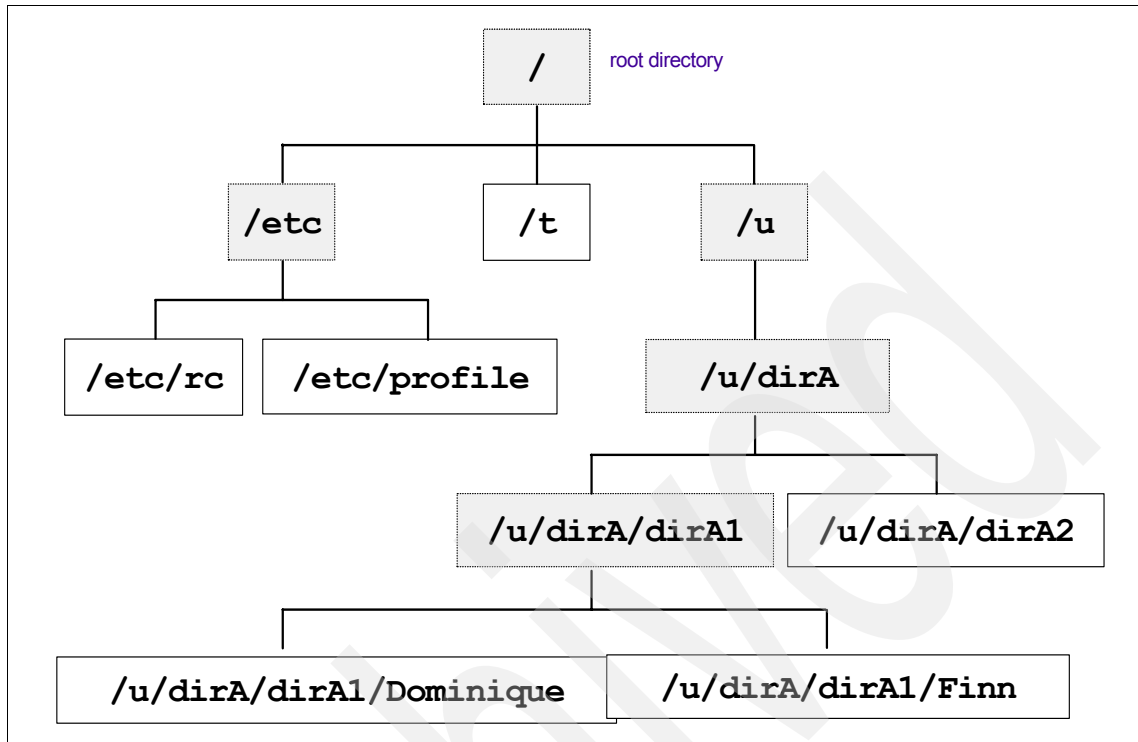


Figure 1-1 Hierarchical file system

The physical file system

The physical file system, as the name implies, is in charge of the physical arrangement of data and control information about the physical media. The physical file system operates with control blocks such as the *superblock*, *inodes*, and *data blocks*. The superblock holds the control information for the system. Inodes contain similar information for individual files. The data blocks hold the data that makes up the information in the files.

1.2.8 The /etc directory

Note that the /etc directory has a particular use in UNIX systems as, by convention, it is expected to host the parameter files used by either the system or applications.

1.2.9 Daemons

A UNIX *daemon* is a program intended to run continuously and to wait for specific service requests. As the requests arrive, the daemon program forwards them to processes that it creates as appropriate.

1.3 The UNIX security model

UNIX implements a security model to identify and authenticate individual users and to determine their access privileges when it comes to accessing the following resources:

- ▶ Directories
- ▶ Files
- ▶ Interprocess communications (IPC)

1.3.1 Accessing UNIX

To access UNIX interactively, the user has to *log in* to the same specific *user account*. To do so, they use the *rlogin* (remote login) or *telnet* services. *rlogin* and *telnet* are similar except that *login* allows the access request to originate from a trusted host and does not in that case require a password. The use of trusted hosts is a rather common practice with UNIX services and is obviously a potential security exposure. Security-minded organization would rather use services that always proceed with end user authentication.

UNIX is also case-sensitive when processing user ID and password, therefore uppercase characters used in the user ID or password are different from the same characters in lowercase.

1.3.2 UNIX users and groups

A *user* is the term used to describe an individual entity that requests work to be performed in a computer system. A *group* is the term used to describe a list of users who are associated together by system administration according to administrative or functional reasons.

In a UNIX system, users are identified with a unique user name and a numeric value: the *UID* (see the following section). Likewise, groups of users are also identified with a numeric value: the *GID*. Although it is technically feasible to share the same UID between different users, or the same GID between different groups, it is generally strongly discouraged to do so.

The user is also associated to a password value and, in most of the UNIX system, this value is stored in a file named `/etc/passwd`. Therefore, this file requires very special attention regarding its protection against unauthorized access.

UIDs

The user name is an easy-to-remember word, while UID is a number usually between 0 and 65,535, that is, a 16-bit UID value (some UNIX versions also support 32-bit UIDs). The UID is used internally to the system to represent the user's identity.

Important:

- ▶ UID(0) has a special meaning and is reserved for the so-called *superuser*.
- ▶ UNIX access control privileges are assigned to the UID, not the user name.

The UNIX superuser

The UNIX superuser is a privileged user who has unrestricted access to the whole system, that is, all commands and all resources regardless of whatever the setup of access permission is for these resources. The superuser status is granted to all users with a UID(0). From the historical standpoint, there is a convention that the user name for the superuser account is *root*. Do not confuse the term *root* here with the *root subdirectory* in the file system—they are unrelated.

A superuser, or root account is mostly intended to be in charge of system administration. A superuser can:

- ▶ Read, write, and execute all files and directories, regardless of their permission settings
- ▶ Change permissions for any file or directory
- ▶ Add, remove, or update any UNIX user group
- ▶ Mount or unmount any files system
- ▶ Access any device
- ▶ Change the priority of any task running within the system
- ▶ Change the UID associated to a process without providing authentication data for the new UID
- ▶ Start up and shut down the system

GIDs

Each UNIX user is also associated to a group of users. A UNIX group is represented in two ways: A group name and a GID. Group name is an easy-to-remember word, while GID is a number. This information might be stored in the file `/etc/group`. GID is typically a number between 0 and 65,535, where 0 through 99 might be reserved on some UNIX implementations. Some UNIX implementations also support a 32-bit GID.

Important: Unlike UID, GID=0 has no special meaning.

1.3.3 File and directory permissions

Every file or directory in a UNIX file system has three types of *access modes*. Users or groups are granted none, one, or many of these access modes. They are:

- read [r]** A user who has read permission for a file may look at its contents or make a copy of it. For a directory, read permission enables a user to see the file names in that directory.
- write [w]** A user who has write permission for a file can alter or remove the contents of that file. For a directory, the user can create and delete files located in that directory.
- execute [x]** A user who has execute permission for a file can cause the contents of that file to be executed (provided that it is executable code). For a directory, execute permission allows a user to *traverse* the directory to access files or subdirectories.

These access mode permissions are specified three times for each file or directory, for each one of these three possible accessing entities:

- ▶ The owner's of the file or directory. That is whether the owner has read, write, or execute access.
- ▶ The group that owns the file or directory (be aware that this group can be a different group from the owner's group).
- ▶ The other users, that is, users who are not owner of the file or directory and are not member of the owning group.

As already mentioned, these permissions are associated to a UID or GID, not a user name or group name.

Figure 1-2 shows how permission bits are often referred to by their octal representation. For example, if a file is to be updated only by its owner, while

others are allowed to read/execute it, then the octal permission setting is 755 (owner = rwx = 4+2+1 = 7; group = r-x = 4+0+1 = 5; other = r-x = 4+0+1 = 5).

0	---	No access
1	--x	Execute-only
2	-w-	Write-only
3	-wx	Write and execute
4	r--	Read-only
5	r-x	Read and execute
6	rw-	Read and write
7	rwX	Read, write and execute

Bit values
xxx
421

Permission bit examples:

700	owner (7=rwx)	group (0=---)	other (0=---)
755	owner (7=rwx)	group (5=r-x)	other (5=r-x)

Figure 1-2 Octal representation of permissions

1.4 The z/OS UNIX System Services history

In the 1980s, IBM mainframe customers requirements began to surface as the need to leverage their organization investment in UNIX systems and applications skills using a technology exhibiting an industry-proven superior quality of services. Open standards compliance was also emerging as a core strategy for integrating any platform seamlessly into distributed environments and to increase one platform's applications portfolio.

In 1991, IBM decided that MVS should incorporate support of the UNIX user and application interfaces, therefore making the UNIX environment model part of the IBM mainframe offering strategy.

The first implementation was known as MVS OpenEdition (or OE, or OMVS), then it became OS/390 UNIX System Services, and finally z/OS UNIX System Services, as we know it today. The main steps of the evolution were:

- ▶ In 1994, IBM MVS/ESA™ V4 was hosting the first version of Open Edition, which was an optional feature of MVS.
- ▶ The same year Open Edition was integrated into MVS V5R1.

- ▶ In **1996**, IBM OS/390® V1R2 received the official UNIX branding. That is, the IBM mainframe proprietary operating system became an official UNIX-compliant system. Figure 1-3 is a graphical view of the standards OS/390 had to conform to get the UNIX branding.
- ▶ In **1997**, the Open Edition kernel was merged into the core operating system, and in **1998** OpenEdition was renamed to UNIX System Services with OS/390 V2R6.

Important: z/OS implements the POSIX-compliant APIs as extensions to MVS system services, that is, the UNIX System Services is a set of MVS architected services that behave, from the application standpoint, as though the system were a UNIX system.

However, note that this implementation, as explained in Chapter 2, “Overview of z/OS UNIX implementation” on page 13, also allows UNIX applications to take advantage of specific z/OS operating system features such as Workload Manager (WLM), System Management Facilities (SMF), Sysplex, and so on, and provide optional (but recommended, and explained in this book) enhancements over the regular UNIX security model.

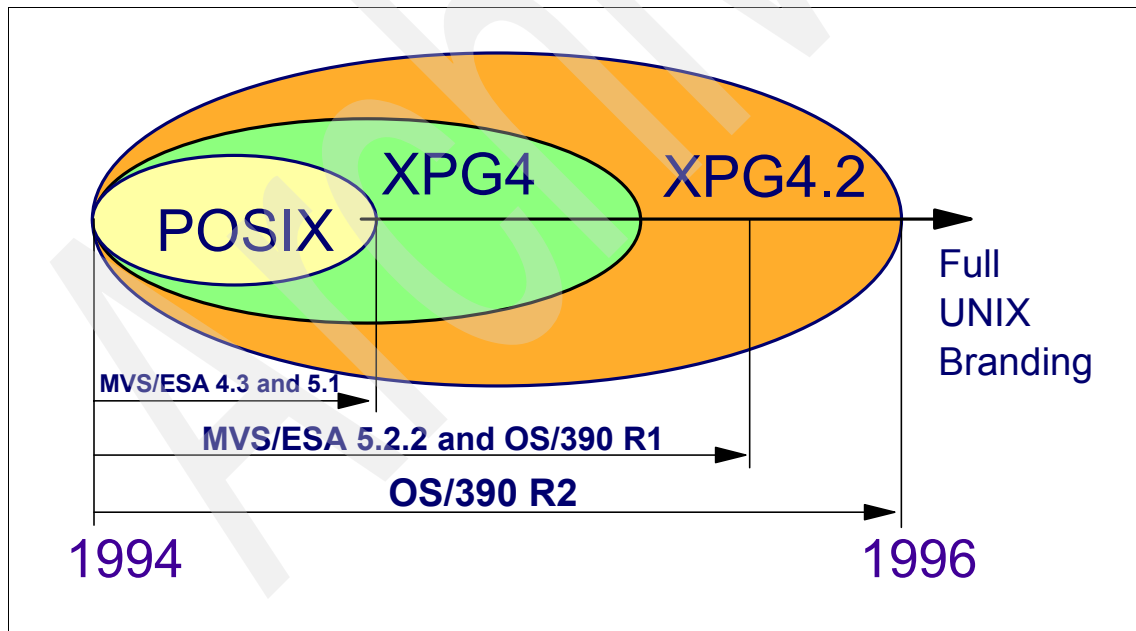


Figure 1-3 OS/390 UNIX branding and standards



Overview of z/OS UNIX implementation

In this chapter, we provide an overview of how UNIX functions are implemented in z/OS, stressing, whenever appropriate, functions with similar purposes in the MVS and z/OS UNIX environments.

2.1 z/OS UNIX System Services fundamentals

The z/OS UNIX System Services provides a UNIX-like operating environment, which is implemented via an extension to the MVS architected services provided by the z/OS operating system.

The z/OS support for UNIX System Services makes two open systems interfaces available to the z/OS operating system users:

- ▶ An application programming interface (API) for applications to request UNIX operating system services. This API is primarily intended for C language, programs executing in the IBM z/OS Language Environment® (LE); but UNIX functions are made available as assembler callable services as well.
- ▶ An interactive z/OS UNIX shell interface.

Figure 2-1 shows the API and interactive shell open systems interfaces and their integration within z/OS. Notice the z/OS UNIX kernel address space that hosts the UNIX System Services main control functions.

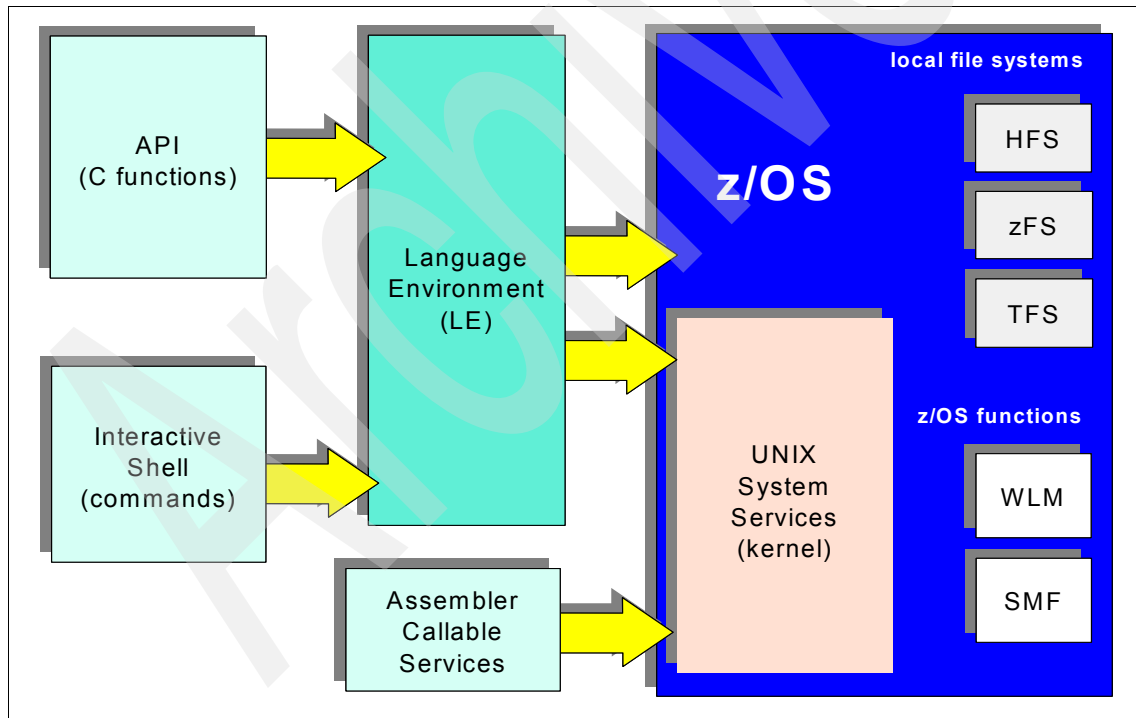


Figure 2-1 z/OS UNIX with open systems interfaces

These APIs are also available in different program execution environments such as batch processing, jobs submission from the Time Sharing Option Extensions (TSO/E) environment, started tasks, or other z/OS application task environment. A z/OS application program can therefore be designed to use the operating system APIs to call for:

- ▶ MVS services only
- ▶ z/OS UNIX services only
- ▶ Both MVS and z/OS UNIX services

The shell interface is available on TSO/E terminals, or through telnet or rlogin client system. A default shell program is provided in z/OS with the capability of overriding the default to give users access to another specific shell program. At the time of the writing of this book, z/OS comes with two shell programs: The z/OS UNIX shell and the tcsh shell. Users can also install their own shell program.

The system tasks originating from the shell are:

- ▶ Programs run by shell users
- ▶ Shell commands and scripts run by shell users
- ▶ Shell commands and scripts run as batch jobs

Note: In order to stress the differences between the z/OS components that implement z/OS UNIX functions and the other components of the system, we sometimes refer to the latter as the system's MVS components or functions.

2.1.1 Dubbing

As already mentioned, z/OS UNIX is made up of MVS architected services that behave as though the operating system is a UNIX system. Therefore, all applications are started as regular MVS address spaces without initially caring whether they have to acquire UNIX process characteristics. The MVS address space is given the characteristics of a UNIX process at the time of the first call to a z/OS UNIX system service. The z/OS address space is then said to be *dubbed* and the system makes necessary updates to control blocks so that from now on the address space is also considered to be a UNIX *process*.

Note that new address spaces or tasks created as a direct consequence of calls by existing UNIX process or thread are systematically dubbed when they are initialized.

Important: In addition to the user having a UID, z/OS UNIX also requires that the UNIX user MVS default group and current connect group have a GID for the dubbing to work.

2.1.2 z/OS UNIX services

This section provides information about the z/OS UNIX services.

System Services

The System Services provide:

- ▶ XPG4 UNIX 1995 and 1998 functions conformance
- ▶ Assembler callable services
- ▶ TSO/E commands to manage the file system
- ▶ A shell environment via ISPF panels

Application Services

Application Services interprets commands from users or shell scripts, and calls z/OS services for their execution. The Application Services provide:

- ▶ A TSO/E command to enter the shell environment
- ▶ A shell environment for developing and running applications
- ▶ Utilities to administer and develop in a UNIX environment
- ▶ The dbx debugger utility environment
- ▶ Support for socket applications
- ▶ rlogin (remote login) and inetd functions
- ▶ Direct telnet based on TCP/IP protocol
- ▶ Support for full-screen applications (curses support)

z/OS and UNIX functional comparison

Table 2-1 provides a functional comparison of some of the basic functions of z/OS and the equivalent UNIX functions as provided by z/OS UNIX.

Table 2-1 z/OS and UNIX functional comparison

Function	z/OS	UNIX
Background work	Submit batch JCL	sh_cmd &
Configuration parameters	SYS1.PARMLIB	/etc
Data management	DFSMS, HSM	tar, cpio, pax
Debug	TSO TEST	dbx
Editor	ISPF option 2	ed, sed, oedit, ishell
Initiate new task	ATTACH, LINK, XCTL	fork(), spawn()
Interactive access	Logon to TSO	telnet/rlogin to sh/tcsh
Job management	SDSF	ps, kill()

Function	z/OS	UNIX
List files	ISPF option 3.4, LISTC	ls
Long running work	Started task (STC)	daemon
Post IPL commands	COMMNDxx	/etc/rc
Power user	RACF OPERATIONS	superuser or root
Primary configuration	IEASYSxx	BPXPRMxx
Primary data index	Master Catalog	root ("/") directory
Procedural language	CLIST, REXX	shell scripts, REXX
Program products	LNKLST	/usr
Resident programs	LPA	sticky bit
System logging	SYSLOG	SYSLOGD
System programs	LNKLST	/bin
Test programs	STEPLIB	/sbin
User data	&SYSUID or &SYSPREF	/u/<username>
User identity	user/group	UID/GID

2.1.3 z/OS UNIX and z/OS features

z/OS UNIX interacts with the elements and features of z/OS that are described in the following sections.

Workload Manager

The Workload Manager (WLM) is a base component of z/OS that is used by the UNIX kernel to create child processes, that is, new address spaces. When programs issue the `fork()` or `spawn()` requests to create a new UNIX process, as shown in Figure 2-2, the BPXAS PROC, installed in SYS1.PROCLIB, is used to provide a new address space.

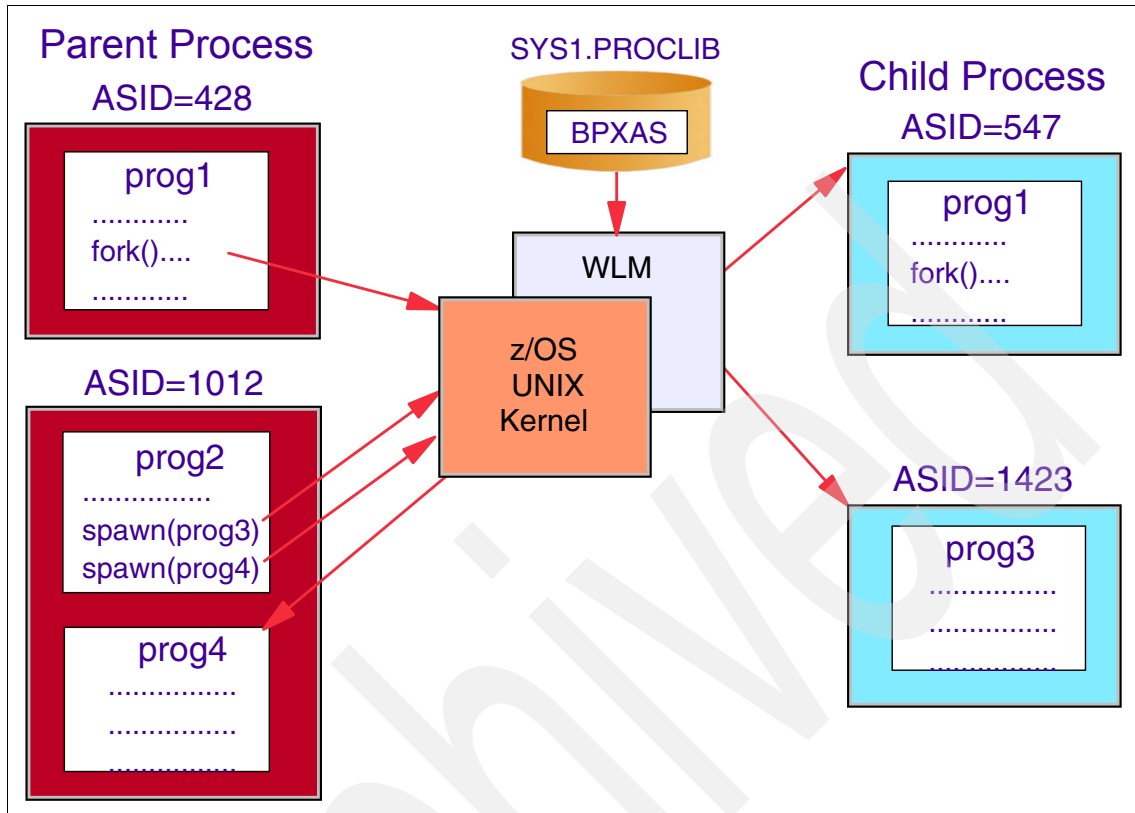


Figure 2-2 Examples of a parent process issuing fork() and spawn()

The types of processes are:

- ▶ User processes, that is, typically applications which are started on a specific user request.
- ▶ Daemon processes, which perform continuous or periodic system-wide functions, such as a Web server. Daemons are programs that are typically started when the operating system is initialized and remain active to perform standard services. In z/OS UNIX, daemons can be started as started tasks (STC). Examples of typical UNIX daemons are:
 - cron, which starts applications at specific times.
 - inetd, which provides service management for a network.
 - rlogind, which starts a user shell session when requested to do so by the **rlogin** command.

A process can have one or more threads. A thread is a single flow of control within a process. Application programmers create multiple threads to structure an application in independent sections that can run in parallel for more efficient use of system resources. In z/OS UNIX, a multi-threaded process is an address space that contains several subtasks.

System Management Facilities

System Management Facilities (SMF), which is a base component of z/OS, collects data for accounting. SMF job and job-step accounting records identify processes by user name, process PID, group GID, and session identifiers. Fields in these records also provide information about resources used by the process. SMF file system records describe file system events such as file open, file close, and file system mount, unmount, quiesce, and unquiesce.

The JWT value in the SMF parmlib SMFPRMxx can be used to specify when to time out an idle address space. SMF/WLM does the tracking.

C/C++

The C/C++ compiler, which is available with z/OS, is needed to compile C code using the `c89` command, or to compile C/C++ code using `cxx`.

Language Environment

The C/C++ runtime library provided with Language Environment (LE) is needed to run a shell command or utility, or any user-provided application program written in C or C++.

Data Facility System Managed Storage

Data Facility System Managed Storage (DFSMS) can be used to manage the data sets used to host the Hierarchical File System (HFS). These data sets make up a file hierarchy that contains directories and files, which can integrate as well remote file systems (that is, using network file system access such as with NFS) mounted within the hierarchy. Note that two types of data sets that can host the z/OS UNIX Hierarchical File System are available: The HFS data sets and VSAM linear data sets for the so-called *zFS* file system.

Security Server

z/OS UNIX requires an external security manager to be accessible through the System Authorization Facility (SAF) interface. The examples that we provide in the rest of this book assume that the IBM Resource Access Control Facility (RACF) external security manager is used, although similar non-IBM products, which provide equivalent functions, can be used.

RACF is delivered in the z/OS Security Server component and is used to hold the z/OS UNIX users and groups registry and to also perform access control and auditing for accesses to the z/OS UNIX resources.

Important: All audited access violations to z/OS UNIX resources are expected to get external security manager messages being issued, as it is the case for MVS resources, on the system console (typically these are the ICH408I messages with RACF).

Note that the examples that we provide in this book apply to RACF and are not expected to be directly usable with other external security managers.

2.1.4 Resource Measurement Facility

IBM Resource Measurement Facility (RMF™) collects data used to report on z/OS UNIX performance. RMF monitors the use of resources in an address space type of OMVS for z/OS address spaces created by fork or spawn callable services, along with the use of resources in an OMVS Kernel Activity report.

System Display and Search Facility

z/OS UNIX shell users can enter TSO/E sessions and use System Display and Search Facility (SDSF) to:

- ▶ Monitor printing
- ▶ Monitor and control a batch job
- ▶ Monitor and control forked address spaces
- ▶ Find out which users are logged on to TSO/E sessions

Time Sharing Options Extensions

One way to enter the shell environment is by using TSO/E. A user logs on to a TSO/E session and then enters the TSO/E OMVS command.

TSO/E also provides z/OS UNIX commands, for example, to logically mount and unmount file systems, create directories in a file system, and copy files to and from z/OS data sets. Users can switch from the shell to their TSO/E session, enter commands or do some editing, and switch back to the shell.

z/OS Communications Server TCP/IP Services

Another way to enter the shell environment is by using rlogin or telnet from a workstation in the TCP/IP network.

User-written socket applications can use TCP/IP Services as a communication vehicle. Both client and server socket applications can use the socket interface to

communicate over the Internet (AF_INET and AF_INET6) and between other socket applications by using local sockets (AF_UNIX). An assembler interface is also provided for those applications that do not use the C/C++ runtime library.

Interactive System Productivity Facility

Users of ISPF can use the ISPF shell environment to create, edit, browse, and perform other functions for files and directories in the HFS.

BookManager READ/MVS

You can invoke the online help facility with the TSO/E OHELP command and view online publications in BookManager® format.

Network File System

Network File System (NFS) enables users to access files on other systems connected to the network.

2.1.5 z/OS UNIX configuration parameters

z/OS UNIX System Services is an environment within the z/OS operating system itself, and has its own runtime parameters defined in the BPXPRMxx member of SYS1.PARMLIB. See *z/OS V1R8.0 MVS Initialization and Tuning Reference*, SA22-7592, for detailed information about the contents of BPXPRMxx.

2.1.6 z/OS UNIX kernel

The z/OS UNIX kernel always starts at the same time that z/OS starts. Depending on the intended use of the z/OS UNIX System Services, the kernel can start in minimum mode or full mode. These modes are explained in *z/OS V1R8.0 UNIX System Services Planning*, GA22-7800.

In this book, we assume that the kernel is started in full mode.

2.1.7 z/OS UNIX file system

Applications in z/OS UNIX get access to data through physical file systems (PFSS), which can be thought of as a set of access method services with proper APIs to write and read data. The underlying physical file systems are defined in the BPXPRMxx member of SYS1.PARMLIB (with the FILESYSTYPE statement) in order to be automatically activated when z/OS UNIX is started.

A PFS is specialized for a type of file system, with file system having a quite broad meaning in UNIX as it can designate data residing on a disk as well as data being transmitted through sockets.

A simplified description of z/OS UNIX System Services and PFS is given in Figure 2-3 on page 23. The following are all PFSs:

- Hierarchical File System (HFS)

z/OS UNIX files are organized in a HFS, as in other UNIX systems. Files are members of a directory, and each directory is in turn a member of another directory at a higher level, the highest level being the *root* directory. The HFS is implemented in z/OS using HFS or zFS data sets.

- Network File System (NFS)

The NFS client on z/OS UNIX can mount a file system, directory, or file from any system in the network that runs an NFS server, within a z/OS UNIX user's directory. The remote files and directories can then be worked on as though they were local z/OS UNIX files and directories.

- Distributed File System (DFS™)

DFS joins the local file systems of several file server machines making the files equally available to all DFS client machines. DFS allows users to access and share files stored on a file server anywhere in the network, without having to consider the physical location of the file.

- Temporary File System (TFS)

The TFS is an in-memory physical file system that delivers high-speed access to data. A TFS is usually mounted at the /tmp directory, therefore it can be used as a high-speed file system for temporary files.

- Pipe

A program creates a pipe with the pipe() function. A pipe typically is written with data by one process to be read by another process. The two ends of a pipe can also be used in a single program task. A pipe does not have a name in the file system, and it vanishes when the last process using it closes it.

- Socket

A program creates a socket with the socket() function. A socket is a method of communication between two processes that allows communication in two directions, in contrast to pipes, which allow communication in only one direction. The processes using a socket can be on the same system or on different systems in the same network.

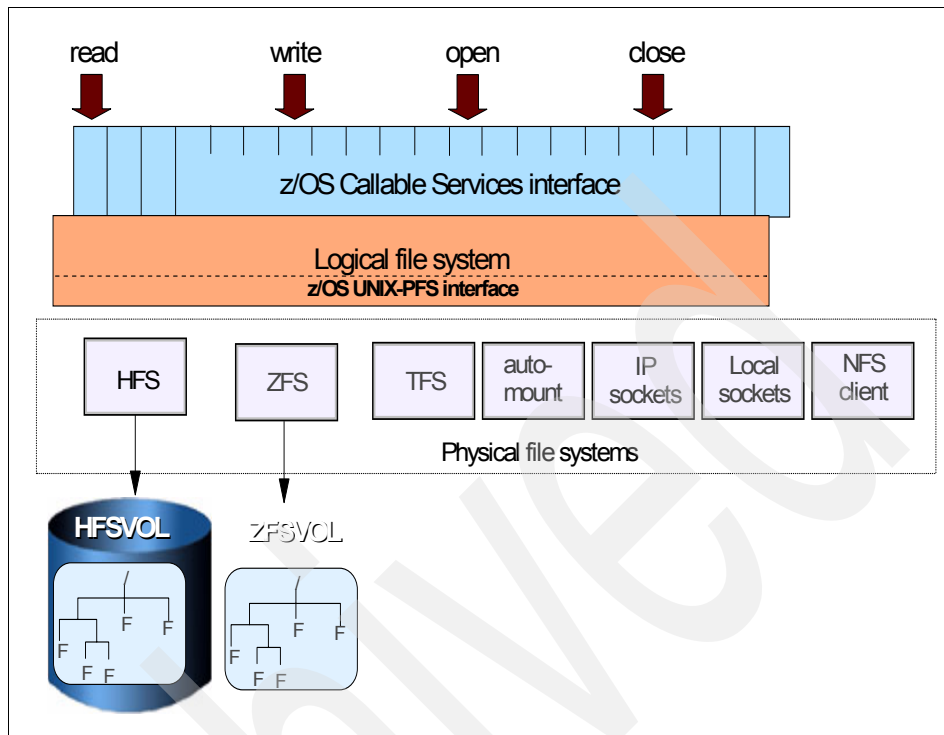


Figure 2-3 z/OS UNIX physical file systems

File system hierarchy and data sets

For some PFS, the files that are managed are to be visible in a file hierarchy. This is the case for the HFS and zFS file systems where data is actually stored in HFS or zFS data sets, in the MVS sense of data repository, which are *mounted* as subsets of the file hierarchy, and are managed by the DFSMS component of z/OS. Figure 2-4 on page 24 shows a hierarchical file system made of three different HFS data sets.

Note: The HFS or zFS data sets can be thought of as *containers* that contain the UNIX file system files and directories. Appropriate MVS protection must be applied to the HFS and zFS data sets themselves, which are accessed with the kernel address space identity. UNIX protection must be applied to the individual files and directories, hosted by the data sets, which are accessed with UNIX users' identities.

Starting from the top of the file hierarchy, that is the root, the root file system is the first file system to be mounted. Subsequent file systems can be mounted on any directory within the root file system or on a directory within any already

mounted file system. Mounting a file system is performed using a **mount** command that can be issued from the BPXPRMxx parmlib member (that is issued at z/OS UNIX startup, as it is the case for the root file system), by a user through ISHELL, by the TSO/E MOUNT command, by automount, or by a program using the mount() function.

Note: The root file system has to be a z/OS UNIX HFS data set. Other file systems mounted under the root can be either HFS or zFS data sets.

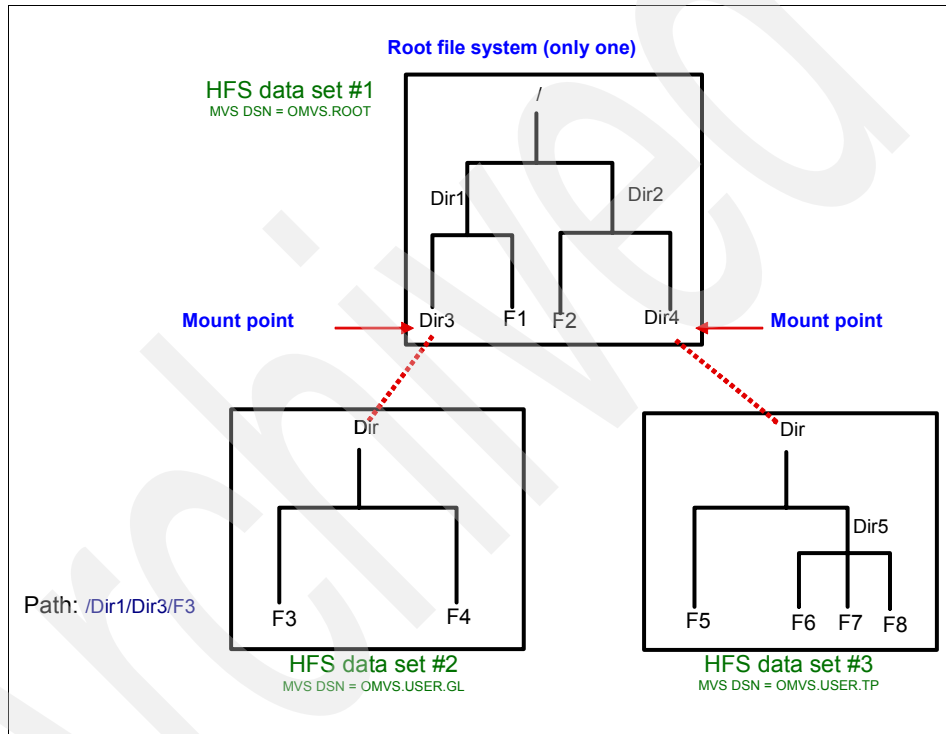


Figure 2-4 Hierarchical file systems and mount points

Symbolic, external, and hard links

Users can establish aliasing in the file system by symbolic or external links.

- ▶ A symbolic link is a file that contains the path name for another file, in essence a reference to the original file. Only the original path name is the real name of the original file. You can create a symbolic link to a file or a directory. In the z/OS UNIX HFS, /etc, /tmp, /dev, and /var are usually symbolic links that contain references to the real path name of these directories.
- ▶ An external link is a type of symbolic link that points to an object outside of the file system. Typically, it contains the name of a z/OS data set.
- ▶ A hard link is an additional name for an existing file. Only one physical file exists, but it can have multiple names represented by hard links. As an example, by establishing a hard link to the file /u/dominique/projects, the path name /u/patrick/projects can point to the same file. You cannot create a hard link to a directory, and you cannot create a hard link to a file on a different mounted file system.

File security packet

Each z/OS UNIX file and directory has a set of information called the *file security packet* (FSP) associated with it to maintain access control permissions and other relevant information. The FSP is created when a file or directory is created, and is kept, with the file or directory, in the file system until the file/directory is deleted, at which time the FSP is also deleted. Figure 2-5 shows the structure of the FSP, which is discussed in details in Chapter 6, “z/OS UNIX files security” on page 73.

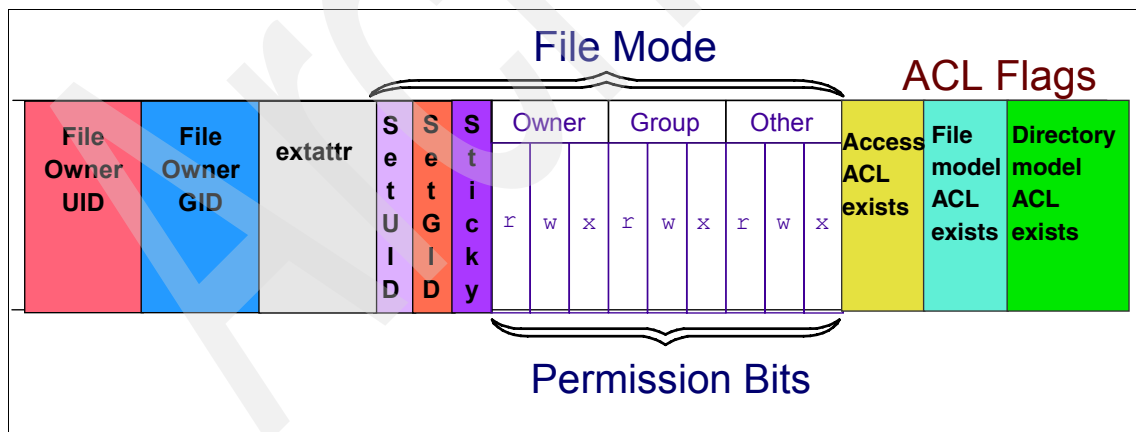


Figure 2-5 File security packet

Important: The FSP can be considered as meta-data that is always associated with the file. This is the case when a file is exported by a z/OS UNIX utility such as the TAR archiving utility: The file security packet is exported with the file data.

With such an implementation, although RACF is used to control access to files or directories, there are no RACF profiles to define access permissions for UNIX files or directories resources.

2.2 Securing the z/OS UNIX environment

The z/OS UNIX environment is established with a set of z/OS resources. In this section, we describe how to set up protection for these resources. We begin with a description of the two started procedures that are used to initialize the z/OS UNIX kernel:

- ▶ OMVS
- ▶ BPXOINIT

Then we describe the procedure that is used to invoke WLM for the creation of z/OS UNIX process address space:

- ▶ BPXAS

2.2.1 z/OS UNIX address spaces

The OMVS and BPXOINIT started procedures are invoked to initialize the z/OS UNIX System Services environment, as shown in Figure 2-6 on page 28.

The OMVS STC

The OMVS address space runs a program that initializes the kernel. The STARTUP_PROC statement in the BPXPRMxx member of SYS1.PARMLIB specifies the name of the cataloged procedure that resides in SYS1.PROCLIB. It is strongly recommended that this procedure name remain its default value of OMVS; changing it is likely to cause some impact with related functions such as TCP/IP.

This procedure requires an entry in the Started Task Table or a profile in the RACF STARTED class. It must be given an MVS user ID with a UNIX UID that belongs to a group with a UNIX GID. In Chapter 3, “z/OS UNIX users and groups identity management” on page 31, we explain in detail how to allocate UID and GID. For the time being, we just remind you about the RACF commands to be issued to assign a UNIX user and group to the OMVS cataloged procedure.

Note that the very commonly used group name and user name for the OMVS started procedure are OMVSGRP and OMVSKERN. Note also that the OMVS user ID must be given UID(0).

```
ADDGROUP <group_name> OMVS(GID(<an_number>))
```

```
ADDUSER <MVS_userID> DFLTGRP(the_default_group_name) OMVS(UID(0)  
HOME('/') PROGRAM('/bin/sh')) NOPASSWORD
```

Note: Here we use the NOPASSWORD user attribute resulting in defining a *protected* RACF user, that is, a user ID that is used to log on to the system and cannot be revoked by incorrect password attempts.

To define the cataloged procedure in the STARTED class of RACF profiles:

```
RDEFINE STARTED OMVS.* STDATA(USER(<MVS_userID>) GROUP(group_name)  
TRUSTED(YES))
```

Note: The kernel address space is to access resources such as the HFS or zFS data sets. You must decide whether to mark OMVS (the kernel) trusted for access. Making the kernel trusted is useful for giving the kernel access to any local data set that it wants to mount. If you do not mark the kernel TRUSTED for local access, set up profiles so that the kernel user ID has access to any local data set that it needs to mount.

For instance, if the installation naming convention gives OMVS as the high-level qualifier for HFS or zFS data sets, the user ID of the OMVS started procedure must be permitted to this high-level qualifier:

```
PE 'OMVS.**' ID(<MVS_userID>) ACC(UPDATE)
```

If the procedure runs not TRUSTED, the MVS_userID must also be permitted in READ to SYS1.PARMLIB.

In the rest of the section, we assume that you elected to assign the user ID OMVSKERN and the group OMVSGRP to the OMVS and BPXOINIT address spaces.

The BPXOINIT STC

BPXOINIT is the procedure that runs the z/OS UNIX initialization process and is called at z/OS startup by the OMVS started procedure.

The BPXOINIT address space gets the process ID (PID) “1”. This is the parent of /etc/rc, that is other processes that users have set up to be started at initialization time. This task is also the parent of any z/OS address space that is explicitly

dubbed, which is not created by fork() or spawn(). Therefore, TSO/E commands and batch jobs that invoke the z/OS UNIX System Services have a parent PID of “1”.

The BPXOINIT started task shares the MVS group and user ID, and therefore the same UID and GID, with the OMVS catalogued procedure.

Note: The BPXOINIT started procedure is recommended to be run not TRUSTED, and therefore the user ID OMVS must be permitted to the resources accessed by BPXOINIT.

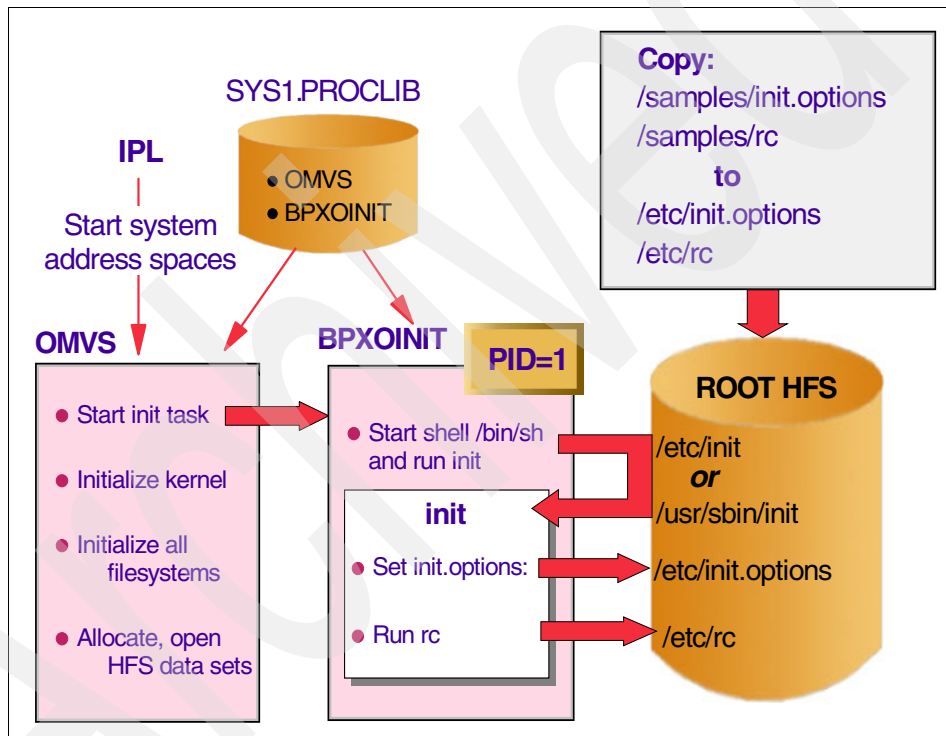


Figure 2-6 The z/OS UNIX initialization process

The files associated with the system initialization program `/usr/sbin/init` are as follows:

<code>/bin/sh</code>	Default shell that <code>/usr/sbin/init</code> invokes to execute <code>/etc/rc</code> or another shell script specified in the <code>/etc/init.options</code> file
<code>/etc/init.options</code>	Initialization options file, which is read by <code>/usr/sbin/init</code>
<code>/etc/rc</code>	Default initialization shell script
<code>/etc/log</code>	The file that output is written to
Other utilities	Services that are called by the initialization shell script

Important: Because the processes created by `/usr/sbin/init` inherit the GID of `BPXOINIT`, do not permit the `OMVSGRP` to any `MVS` resources, unless programs you start using `/etc/rc` need to be permitted to these resources.

Note that any programs forked by `/etc/rc` receive their authority from the user ID assigned to the `BPXOINIT` process. Use the same user ID for `BPXOINIT` as you assigned to the kernel (`OMVSKERN`). The `BPXOINIT` process and any programs forked by the kernel's descendants will have therefore superuser authority. This is essential for the successful execution of initialization tasks run from `/etc/rc`.

The BPXAS STC

The `BPXAS` procedure is started by `WLM` when a `z/OS UNIX` program uses the `fork()` or `spawn()` function, or the equivalent callable services. The `BPXAS` procedure found in `SYS1.PROCLIB` is used to provide a new address space. For a fork, the system copies one process, called the parent process, into a new process, called the child process. Then it places the child process in a new address space. The forked address space is provided by `WLM` using the `BPXAS` procedure.

The `BPXAS` procedure shares the same `MVS_userID` and `group_name` as the `OMVS` started task and is recommended to be run not `TRUSTED`.

2.2.2 HFS and zFS data sets

`RACF` profiles that protect the `HFS` and `zFS` data sets are created with `UACC(NONE)`. They do not need any permission in their access list if the `OMVS STC`, which initializes the kernel address space, has been given the `TRUSTED` attribute. Otherwise, the `OMVS` address space user ID (`OMVSKERN` in our examples) must have `UPDATE` access to these data sets.

2.2.3 Protecting the BPXPRMxx member

This falls under the expected protection of the SYS1.PARMLIB data set, using a profile in the RACF DATASET class of resources, where permissions are usually given to system programmers only.

2.2.4 Protecting z/OS UNIX related operator commands

It is recommended to protect access to operator commands that can be used to dynamically change parameter values in the BPXPRMxx member. These are the SET OMVS and SETOMVS commands, which can be protected using the following profiles in the OPERCMDS class of profiles:

- ▶ RDEFINE OPERCMDS MVS.SET.OMVS
- ▶ RDEFINE OPERCMDS MVS.SETOMVS.OMVS
- ▶ PE MVS.SETOMVS.OMVS CLASS(OPERCMDS) ID(group) ACC(READ)
- ▶ PE MVS.SET.OMVS CLASS(OPERCMDS) ID(group) ACC(READ)

2.3 Applications security: UNIX security and z/OS UNIX security

In Chapter 5, “The z/OS UNIX security model” on page 51, we explain the use of the BPX.SERVER and BPX.DAEMON profiles in the FACILITY class. If none of these profiles are defined, the system is said to have UNIX-level security. In this case, the system is less secure than what can be achieved with z/OS UNIX. This level of security is for installations where superuser authority has been granted to system programmers. These individuals already have permission to access critical MVS data sets such as PARMLIB, PROCLIB, and LINKLIB. These system programmers have total authority over a system.

2.4 RACF AIM

Application Identity Mapping (AIM) is a set of fields in the user profile intended to assist the mapping of a RACF user ID to a UNIX, Lotus® Notes®, or Novell directory identity. The RACF database is reorganized with an alias index when brought to the so-called AIM Stage 3 using the RACF IRRIRA00 utility. The current AIM stage of the RACF database is meaningful in the z/OS UNIX context because, if Stage 2 or Stage 3 of identity mapping has not been reached, there is a potential performance issue with mapping UIDs to user IDs both ways. It is recommended that either Stage 2 or Stage 3 of AIM be reached with RACF as soon as possible.

z/OS UNIX users and groups identity management

In this chapter, we address:

- ▶ The z/OS implementation of the UNIX user identity and the related system identification and authentication processes.
- ▶ The administrative functions available in Resource Access Control Facility (RACF) to assign UNIX identities to users and groups.

Note the following conventions that are adopted in the rest of the book:

- ▶ The *userID* pertains to the MVS identity
- ▶ The *username* is the user's name in the context of the UNIX model of operating system.

3.1 User identification and authentication in z/OS UNIX

z/OS UNIX abides with the principles of UNIX identification and authentication, as described in 1.3, “The UNIX security model” on page 8:

- ▶ The users identify themselves to the system using a user name.
- ▶ After the user is authenticated using the user name, the user initiated processes or threads are assigned the user’s UID. This UID is further used to determine the user’s privileges.
- ▶ Users are also associated with groups. Being part of a group grants the privileges given to the group entity.

3.1.1 User identity implementation

z/OS UNIX requires an external security manager to be operating in the z/OS instance. As already mentioned, we are assuming in the rest of the book that IBM RACF is this external security manager.

The z/OS UNIX identity is an extension given to the MVS identity of the user:

- ▶ All system users have to be registered as MVS users in the RACF database. That is, they are given a RACF USER profile with a RACF user ID and a password (unless they are defined with the NOPASSWORD attribute in the USER profile).
- ▶ Their RACF user ID is their UNIX user name and they are allocated a UNIX UID by the system administrator if they are to use the UNIX System Services. The corollary to this is that the UNIX user name is guaranteed to be unique, as the RACF user ID is. We will see below that the individual UID does not have to be unique and can be shared, although not recommended, between different users.
- ▶ If they are given a UID, then their MVS user’s default and current connect group must also be given a GID.

Note: There is a divergence in the way that we specify the user name with non-z/OS UNIX systems in that RACF is always folding the user name to uppercase, while UNIX uses mixed case user names. z/OS offers the USERIDALIASTABLE mechanism if there is a strict need to deal with mixed case user IDs. For more details about the USERIDALIASTABLE, see *z/OS V1R8.0 UNIX System Services Planning, GA22-7800*.

This implementation results in the z/OS UNIX user tasks that run in the system being given a dual identity: They do have an MVS user ID that is used to

determine what their access rights for MVS resources are, as specified in the RACF resources profiles, and they have a UNIX UID used for access control to z/OS UNIX resources such as files and directories. This is represented in Figure 3-1.

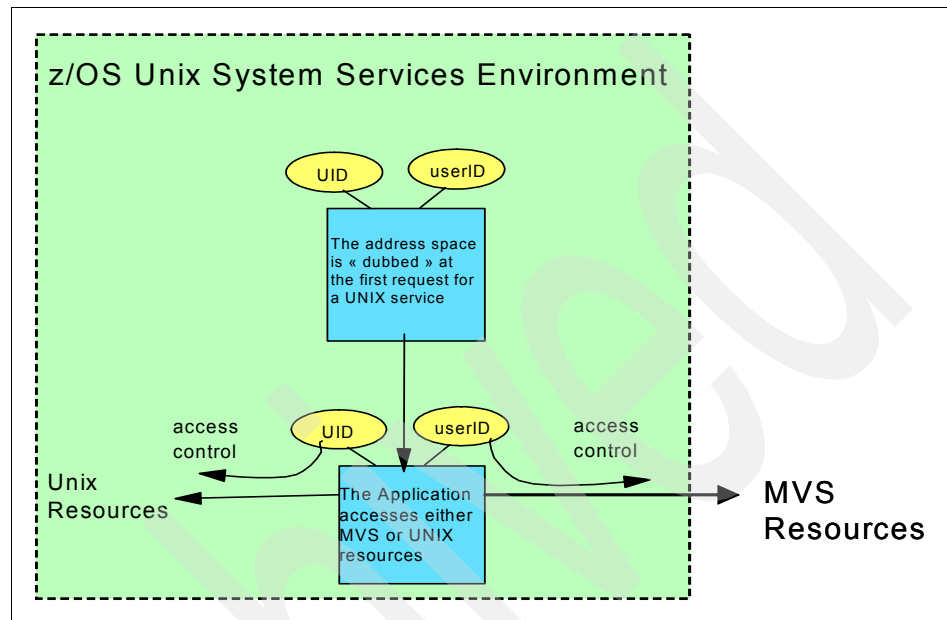


Figure 3-1 The z/OS UNIX user dual identity and access rights

3.1.2 User authentication

z/OS UNIX users are authenticated by RACF through the System Authorization Facility (SAF) interface, by using their MVS user ID (which equates their UNIX user name) and their MVS password, which is stored in the USER profile.

Passtickets can also be used as an alternative to passwords. A passticket is a 64-bit value that is cryptographically composed by a passticket generation software. The passticket is accepted as authentication data by RACF under the condition that the passticket generator program and RACF share a common secret cryptographic key and proper profiles have been defined in the RACF database.

z/OS UNIX applications request RACF to validate a password or passticket by using the initACEE RACF callable service. The initACEE RACF callable service supports the use of mixed case passwords if the RACF MIXEDCASE option is in effect (mixed case passwords are supported starting with z/OS V1R7). initACEE also supports, beginning with z/OS V1R8, the use of *password phrase* as an

alternative to passwords. (Be aware though that, at the time of the writing of this book, there is not yet any IBM product that uses password phrase.)

Reminder: Support is provided in z/OS for UNIX applications that accept X.509 V3 digital certificates or Kerberos tickets as means of authentication. RACF is not involved in the authentication process in that case, however, it can map a RACF user ID to the authenticated digital certificate or the Kerberos ticket. This mapping is performed by the R_usermap RACF callable service. As a result of this mapping, the z/OS UNIX user is allocated the UID associated with the mapped-to RACF user ID.

3.2 The UID and GID in z/OS UNIX

In z/OS UNIX, the following information is required for each UNIX user:

- ▶ A UID, which is a 32-bit number between 0 and 2,147,483,647, and which identifies the user as a z/OS UNIX user. This information is specified in the OMVS segment of the RACF USER profile or in the profile BPX.DEFAULT.USER, in the FACILITY class. See the discussion about default UID in 3.3, “Default UID and GID” on page 38.
- ▶ A GID, which is a number between 0 and 2,147,483,647, and which identifies a z/OS UNIX group of users. This information is specified in the OMVS segment of the RACF GROUP profile the user belongs to.

Before assigning UID or GID values the following must be considered: The POSIX 1003.1 standard defines formats for pax, tar, and cpio archives that limit the UIDs and GIDs that can be stored to the following maximum of 16,777,216. Values larger than these will not be properly restored for tar and cpio formatted archives. For USTAR formatted archives, because the user and group names are also stored in the archive, the correct UID and GID will be restored only if the same name is defined on the target system as well.

3.2.1 The OMVS segment in the RACF USER profile

The OMVS segment is used to specify the characteristics of the UNIX user. Typically, there are three basic fields in the OMVS segment: The UID, the HOME, and the PROGRAM fields. There are other fields to be used in specific cases called the *individual limits*. We explain below the contents of these fields.

- ▶ **UID:** This is the numeric UID value assigned to the user.
- ▶ **HOME:** This is the path name of a directory in the file system that automatically becomes the directory the user is in when he or she enters the UNIX shell.

Note that specifying a HOME directory in the OMVS segment does not give the user any access permission to this directory. The directory access remains controlled by the directory permission bits and the optional access control lists (ACLs).

- ▶ **PROGRAM:** This is the path name of the shell program that is started when the user begins a UNIX shell session. Current values for the PROGRAM field are /bin/sh for the z/OS UNIX shell and /bin/tcsh for the tcsh shell. The PROGRAM value is also used to start the shell created as a result of the execution of the **rlogin**, **su**, or **newgrp** commands.

The individual limits fields

The system resource available for z/OS UNIX users is limited as specified in the BPXPRMxx member of SYS1.PARMLIB. However, the RACF administrator can assign individual limits to a specific user in optional fields of the OMVS segment in the USER profile. These optional fields of the OMVS user segment are individual limits that override, for the subject MVS user ID, the general limits given in the BPXPRMxx member. They are:

- ▶ **ASSIZEMAX** for the maximum address space size allocated to the user
- ▶ **CPUTIMEMAX** for the maximum CPU time
- ▶ **FILEPROCMAx** for the maximum number of files per process
- ▶ **MEMLIMIT** for the maximum number of bytes of non-shared memory per user
- ▶ **MMAPAREAMAX** for the maximum memory map size
- ▶ **PROCUSERMAX** for the maximum number of processes per UID
- ▶ **SHMEMMAX** for the maximum number of bytes of shared memory per user
- ▶ **THREADSMAX** for the maximum number of threads per process

Note: The individual limit is assigned to the specific MVS user ID and its associated UID. Another MVS user ID, even sharing the same UID, will have its own individual limits or none.

Here is an example of a z/OS UNIX USER profile definition in RACF:

```
ALTUSER <MVS_userID> OMVS(UID(<number>))PROGRAM(/bin/sh)
HOME('/u/username') THREADSMAX(value) MMAPAREAMAX(value)
MEMLIMIT(value) ASSIZEMAX(value) CPUTIMEMAX(value) FILEPROCMAx (value)
PROCUSERMAX(value))
```

Important: RACF administration for the user's OMVS segment: As for other fields in the USER profile, it is possible to authorize specific users or groups to manage the OMVS segment fields. This permission has to be very carefully granted as it might allow, for instance, users to change their UID to UID(0), or more generally to acquire user characteristics not agreed on by the installation security policy. A good practice is first to define the following profile:

```
RDEFINE FIELD USER.OMVS.* UACC(NONE)
```

Then permit the specific users or groups that need to access the resource:

```
PERMIT USER.OMVS.<field_name> CLASS(FIELD) ID(<user_or_group_ID>)  
ACCESS(<access_type>)
```

Not specifying the OMVS segment or some of its fields

The following sections describe the consequences of not specifying the OMVS segment or some of its fields.

Not specifying the OMVS segment

- ▶ The user is not to get any z/OS UNIX user characteristics and any task running with the user identity will not be *dubbed*, and therefore unable to invoke the z/OS UNIX System Services. Unless the BPX.DEFAULT.USER profile is defined.
- ▶ If there is a BPX.DEFAULT.USER profile defined, any task running with the user identity gets the z/OS UNIX user characteristics as specified in the BPX.DEFAULT.USER profile and is dubbed upon invocation of any z/OS UNIX System Service.

See the explanation for the BPX.DEFAULT.USER profile in 3.3, “Default UID and GID” on page 38.

Specifying the OMVS segment without the UID field

This can be achieved specifying, for instance, OMVS(NOUID) when defining the USER profile.

- ▶ The user will never get the characteristics of a z/OS UNIX user. Any task running under this user identity will never be dubbed. This is the method to use if, for any reason, you want to ensure that an MVS user is never to be a z/OS UNIX user (as this also prevents use of the BPX.DEFAULT.USER profile).

Specifying the OMVS segment without the HOME field

- ▶ If a home directory is not specified in the OMVS Segment, the root (/) directory is then assumed to be the user's home directory.

Specifying the OMVS segment without the PROGRAM field

- ▶ No PROGRAM field in the OMVS segment will default to the /bin/sh program, that is the z/OS UNIX shell, with the FSUM2386 warning message when invoking the shell.

3.2.2 RACF group and z/OS UNIX

A z/OS UNIX user has to be member of at least one z/OS UNIX group. There is a one-to-one mapping between an MVS group and a z/OS UNIX group: The MVS user with a UID belongs to an MVS group, which should also be a z/OS UNIX group. An MVS group becomes a z/OS UNIX group by specifying an OMVS segment with a GID in the GROUP profile. z/OS UNIX groups are given access permissions to z/OS UNIX resources on the basis of their GID as MVS groups are given access to MVS resources. We show an example of z/OS UNIX user and group definition in Figure 3-2.

User profile								
Userid	Default Group	Connect Groups		TSO	DFP	OMVS		
SMITH	PROG1	PROG1	PROG2	UID	Home	Program
						15	/u/smith	/bin/sh

Group profile						
Groupid	Superior Group	Connected Users				OMVS GID
PROG1	PROGR	SMITH	BROWN	25

Group profile (no OMVS segment)						
Groupid	Superior Group	Connected Users				
PROG2	PROGR	SMITH	WHITE	

Figure 3-2 RACF user and group profiles

In this example, we have three RACF profiles:

- ▶ The first profile is a user profile for TSO/E user ID SMITH, which is connected to two MVS groups, PROG1 and PROG2. SMITH is defined as a z/OS UNIX user because he has a UID specified. The UID of user ID SMITH will be 15.

His home directory is /u/smith and he will enter the z/OS UNIX shell when he issues the OMVS command as the name of the shell, /bin/sh is specified as the PROGRAM name.

- ▶ The second profile is a profile for group PROG1, which is the user ID SMITH's MVS default group. This is a z/OS UNIX group with a GID specified in the OMVS segment. The GID of PROG1 will be 25.
- ▶ The third profile is also a group profile for group PROG2, which the user ID SMITH is also connected to. But this group PROG2 does not have an OMVS segment and therefore is not a z/OS UNIX group.

The UNIX user with user name SMITH and UID 15 is therefore in the UNIX group with GID 25. The RACF commands to create OMVS segments for user SMITH and group PROG1 are:

- ▶ ALTUSER SMITH OMVS(UID(15) HOME(/u/smith) PROGRAM(/bin/sh))
- ▶ ALTGROUP PROG1 OMVS(GID(25))

Important:

- ▶ The default group of a z/OS UNIX user and the current connect group must have a GID defined, to conform with POSIX standards.
- ▶ In most cases the system administrator *inherits* a structure of RACF groups, which has been created to answer specific requirements regarding the MVS population of users. While preparing and maintaining the system for the use of z/OS UNIX System Services, it might appear that this existing structure does not match well, or is even incompatible with the intended distribution of permissions among the UNIX groups. In that case, do not hesitate to create new MVS groups with a GID, that is UNIX groups, for the sole purpose of building the UNIX groups structure that matches your needs.

3.3 Default UID and GID

Not all users and groups can justify the administrative workload for getting an OMVS segment specified and maintained in their RACF profiles. For example:

- ▶ Users who need to use sockets and do not need any other UNIX services. They need a UID (and be a member of a UNIX group) for the sole purpose of invoking the socket functions, as these functions are being implemented as UNIX System Services.
- ▶ Users who want to run multi-threading PL/I programs. PL/I uses some z/OS UNIX kernel services, and the user needs a UID to get dubbed when invoking these services.

- ▶ Users who just want to experiment with the shell and do not have an OMVS segment defined yet.

For these types of needs, the users are not required to have an OMVS segment defined in their USER profile and can get the benefit of a default UID and of a membership to a default UNIX group. The default OMVS segments, which they are to share with other users also without their own OMVS segment, resides in a USER profile and a GROUP RACF profile specifically created to get their OMVS segments “borrowed” from.

The system administrator has to define the BPX.DEFAULT.USER profile in the RACF FACILITY class, and store the names of these *lending* profiles in the application data field of BPX.DEFAULT.USER. In the example below:

```
RDEFINE FACILITY BPX.DEFAULT.USER APPLDATA(OEUDFLT/OEGDFLT)
```

OEUDFLT and OEGDFLT are arbitrary names given to the RACF USER and GROUP profiles that lend their OMVS segment. Users without an OMVS segment inherit all fields of the OMVS segment of the OEUDFLT user (UID, HOME, PROGRAM, and all other fields). Assuming, for instance, that OEUDFLT has UID(25) and OEGDFLT has GID(57), all users without an OMVS segment who invoke a z/OS UNIX System Service get dubbed with a UID(25) and a membership to GID(57).

The access list of the BPX.DEFAULT.USER profile is ignored.

Tips:

- ▶ We recommend that you use a visually recognizable value for the default UID and GID (such as 999999).
- ▶ To prevent the misuse of the default USER profile, it is recommended to give it the NOPASSWORD attribute.
- ▶ If you want to forbid the usage of z/OS UNIX services to a specific RACF user ID, you must create an OMVS segment with parameter NOUID for this user with the following RACF command:

```
ADDUSER MVSONLY OMVS(NOUID)
```

In this case, the MVS user is not entitled to get a default UID from the BPX.DEFAULT.USER profile, because it has an OMVS segment. The missing UID will also prevent dubbing any task that has this user ID.

- ▶ If you expect to have a lot of users running with the default segment, you might want to set the default user’s individual limits higher than the general limits.

A special use of BPX.DEFAULT.USER

When users have an OMVS segment but are connected to a group without a GID, the installation can use the BPX.DEFAULT.USER profile to lend only a GID to those users. In that case, our OEUDFLT USER profile will not have an OMVS segment, or an OMVS segment without a UID, while the OEGDFLT GROUP profile is properly defined with OMVS segment and GID. Of course, this special use of BPX.DEFAULT.USER is exclusive of the regular use that we have explained above.

Note: To prevent the potential misuse of the default OMVS segment, the callable services kill(), pidaffinity(), trace(), and sigqueue() are not supported when running with the default OMVS segment.

Important: Any user who is eligible to use BPX.DEFAULT.USER is a valid target of an identity switch as explained in 5.2.1, “Reminder on z/OS UNIX identity switching” on page 53.

3.4 Shared UID and GID

Important: The functions described in this section are available if you are running with z/OS V1R4 and later and your RACF database is formatted with Application Identity Mapping (AIM) at least at Stage 2. Refer to *z/OS V1R8.0 Security Server RACF System Programmer's Guide*, SA22-7681, for an explanation on AIM stages and how to convert your RACF database to this level.

3.4.1 Automatic prevention of UID sharing

z/OS UNIX allows several users to share the same UID. Although this might provide some advantages such as allowing users to share the same access permissions to z/OS UNIX resources but still have their own unique password, it is commonly agreed that, besides these specific needs, it is mostly a potential source of ambiguity and confusion. Note however that this sharing of UID cannot be avoided between superusers as they all have UID(0).

Care has to be taken, when creating new regular z/OS UNIX users, not to allocate an already assigned UID. This can be done by a manual or semi-automatic tracking of already assigned numeric values; it can also be done automatically by RACF as explained below.

In order to get the RACF automatic assignment of unique UID working, the SHARED.IDS profile must be defined in the UNIXPRIV class:

```
RDEFINE UNIXPRIV SHARED.IDS UACC(NONE)
```

This profile acts as a switch that enables the verification in RACF for already assigned UID or GID. For example, assume that UID 15 and GID 25 are already assigned; trying to assign the same numeric values to a new UID or GID with SHARED.IDS defined results in the following chain of messages:

```
ADDUSER MARCY OMVS(UID(15))
IRR52174I Incorrect UID 15. This value is already in use by BRADY.
ADDGROUP PMMVS OMVS(GID(25))
IRR52174I Incorrect GID 25. This value is already in use by PROG1.
ADDUSER (HARRY MARCY) OMVS(UID(14))IRR52185I The same UID cannot be
assigned to more than one user.
```

Note: Enabling this functionality does not affect pre-existing shared UIDs. To find occurrences of shared UID or GID, you can use the RACF Database Unload utility (IRRDBU00) along with the ICETOOL utility. Tutorial samples of the invocation of ICETOOL are provided in SYS1.SAMPLIB(IRRICE). Refer to *z/OS V1R8.0 Security Server RACF Administrator's Guide*, SA22-7683, for more information about using the IRRDBU00 and ICETOOL utilities.

3.4.2 Allowing assignment of shared UIDs or GIDs

Even if the SHARED.IDS profile is defined, you may still want some UIDs to be shared between new users. You will then need to be a RACF SPECIAL or have READ permission to the SHARED.IDS profile, and this will allow you to specify the SHARED keyword in the ADDUSER, ALTUSER, ADDGROUP, and ALTGROUP commands. The SHARED keyword overrides the SHARED.IDS control as shown below:

```
PERMIT SHARED.IDS CLASS(UNIXPRIV) ID(UNIXGUY) ACCESS(READ)
SETROPTS RACLIST(UNIXPRIV) REFRESH
```

When user ID UNIXGUY has at least READ access to the SHARED.IDS profile, it can then assign the same UID or GID to multiple users or groups, using the SHARED keyword as follows:

```
ADDUSER OMVSKERN OMVS(UID(0) SHARED)
```

In this example, we are assuming that several users have to share UID(0).

3.5 Automatic UID and GID assignment

Note: The use of automatic UID/GID requires the following:

- ▶ The RACF database must be at AIM Stage 2 or 3. If this is not the case, the automatic assignment attempt fails with the following message:
IRR52182I Automatic UID assignment requires application identity mapping to be implemented.
- ▶ If the SHARED.IDS profile is not defined in the UNIXPRIV, an IRR52183I message is issued and the attempt fails with the following message:
IRR52183I Use of automatic UID assignment requires SHARED.IDS to be implemented.

3.5.1 Specifying automatic assignment of UIDs and GIDs

UIDs and GIDs that are not already assigned can be automatically given by RACF to new users or groups when they are created by the RACF administrator.

In order to activate this function, the profile BPX.NEXT.USER has to be defined in the RACF FACILITY class:

```
RDEFINE FACILITY BPX.NEXT.USER APPLDATA(10/10000)
```

After this profile has been defined, you can now have RACF to automatically assign a z/OS UNIX UID or GID by specifying the keywords AUTOUID or AUTOGID in ADDUSER, ALTUSER, ADDGROUP, and ALTGROUP commands:

```
ADDUSER MARCY OMVS(AUTOUID)  
ADDGROUP PMMVS OMVS(AUTOGID)
```

The APPLDATA field of the BPX.NEXT.USER profile consists of two qualifiers separated by a slash ("/"):

- ▶ The left qualifier specifies a starting UID value, or range of UID value (the extreme values are separated by a dash ("-")).
- ▶ The right qualifier specifies a starting GID value, or range of GID value (separated by a dash ("-")).

It is possible to inhibit the automatic assignment of only UID or only GID. To do so, the corresponding qualifier before or after the slash ("/") has to be omitted, and the intended UID or GID has to be specified in the command.

If BPX.NEXT.USER in the FACILITY class is not defined, then an IRR52179I message will be issued and the attempt fails with the following message:

IRR52179I The BPX.NEXT.USER profile must be defined before you can use automatic UID assignment.

Important: RACF can only enforce uniqueness of UIDs and GIDs assigned using the RACF TSO commands, RACF ISPF panels, or the R_admin callable service (IRRSEQ00). RACF cannot enforce uniqueness of UIDs and GIDs that are assigned by programs that invoke the ICHEINTY or RACROUTE macros.

3.5.2 Automatic UID and GID assignment in an RRSF configuration

In a RACF Remote Sharing Facility (RRSF) environment where user updates are kept synchronized across the network, you want to avoid UID/GID collisions when AUTOUID/AUTOGID is used on multiple nodes, that is for instance AUTOUID in one node propagating a value already assigned in another node. This is accomplished by specifying unique ranges of values in the BPX.NEXT.USER APPLDATA for each node.

If you are using RRSF automatic command direction for the FACILITY class, you must use the ONLYAT keyword to alter the BPX.NEXT.USER profile, even when changing the profile on the system on which you are logged on. ONLYAT tells RACF not to propagate the command outbound so that BPX.NEXT.USER profiles on other nodes are not wiped out. Note that the AT keyword is not sufficient, because it will still be subject to propagation. Figure 3-3 shows how ONLYAT is used.

```
RALT FACILITY BPX.NEXT.USER UACC(NONE) APPLDATA('100-5000/NOAUTO')
ONLYAT(NODEA)
RALT FACILITY BPX.NEXT.USER UACC(NONE) APPLDATA('5001-15000/50-500')
ONLYAT(NODEB)
```

Figure 3-3 Fixing the UIDs and GIDs ranges in a RRSF configuration

Archived



z/OS UNIX task identity management

This chapter describes the fundamental mechanisms that z/OS implements for z/OS UNIX processes and threads identity management.

It addresses the nature of the UNIX identity information that is attached to the execution of the z/OS task, its use by the system, and how it is allocated and changed.

4.1 Implementation of the UNIX process and threads concepts

Tasks to be accomplished in a UNIX system can take the form of a UNIX process or a UNIX thread. We explain these two concepts in this section as a preparation for further discussion regarding the user identity that can be assigned to them.

4.1.1 The UNIX process

The UNIX process is an entity that executes a given piece of code, owns the required resources, and is identified using a Process ID (PID).

As already mentioned, daemon processes are programs executed in the background, with superuser authority, waiting for services requests. Typically when a request is received and entitled to be served, the daemon duplicates itself using a `fork()` or `spawn()` function call, as shown in Figure 4-1.

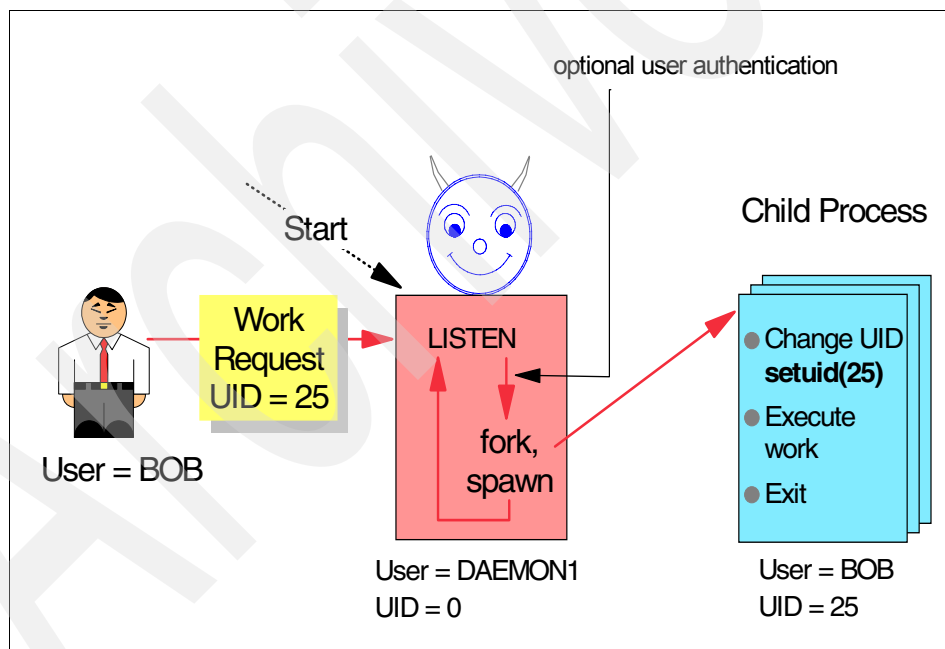


Figure 4-1 UNIX daemon `fork()` or `spawn()`

The duplicated process then changes its identity for the requestor's identity and proceeds to serve the request on behalf of the requestor.

Note that the client's authentication by the daemon process is optional; the superuser authority allows to switch identity without providing authentication data for the target identity. The duplicated *child* process inherits all of the security context of the *parent* process.

The functions called to create the child process can be `fork()`, `spawn()` or `exec()`. `spawn()` is also a POSIX function intended to run more efficiently than `fork()` or `exec()` from the system resource consumption standpoint as there is no duplication of the parent process.

In z/OS UNIX, a process runs in an MVS address space that has been dubbed with the daemon identity characteristics. A process that issues a `fork()` function, creates a new address space, which is a copy of the address space where the program is running. The `fork()` function does a program call to the z/OS UNIX kernel, which then requests Workload Manager (WLM) to create the child process address space. The storage contents of the parent address space are then copied to the child address space.

After the `fork()` function completes, the program in the child address space starts at the same instruction as the program in the parent address space. Control is returned to both programs at the same point with a difference only in the return codes from the `fork()` function:

- ▶ A return code of zero is returned to the child process after a successful `fork()`.
- ▶ The return code for the parent is the child process PID.

Note that any UNIX resources (pipes, sockets, files) accessible via opened file descriptors in the parent are propagated to the new address space. z/OS resources such as DD allocations, cross-memory resources, and ENQ serializations are *not* propagated to the child address space.

4.1.2 The UNIX thread

A server, as opposed to a daemon, can also be defined as a UNIX application servicing client requests, but which uses threading to multitask concurrently executing client requests in the same address space.

As shown in Figure 4-2 on page 48, when a client request is received, the server main process uses the `pthread_create()` call to create a new thread, which executes the client request asynchronously. Thread support improves performance by providing concurrent, asynchronous processing without the requirement to create a new address space and a new process to run the client request.

In the z/OS UNIX implementation, this translates into the main process being the main MVS task in the address space and the threads being started as MVS

subtasks. Although each thread has a dedicated TCB, and can execute a different code subroutine, all the program code to support main task and client threads has to be packaged in a single program shared module.

Each thread (task) can be initialized to run in different security environments.

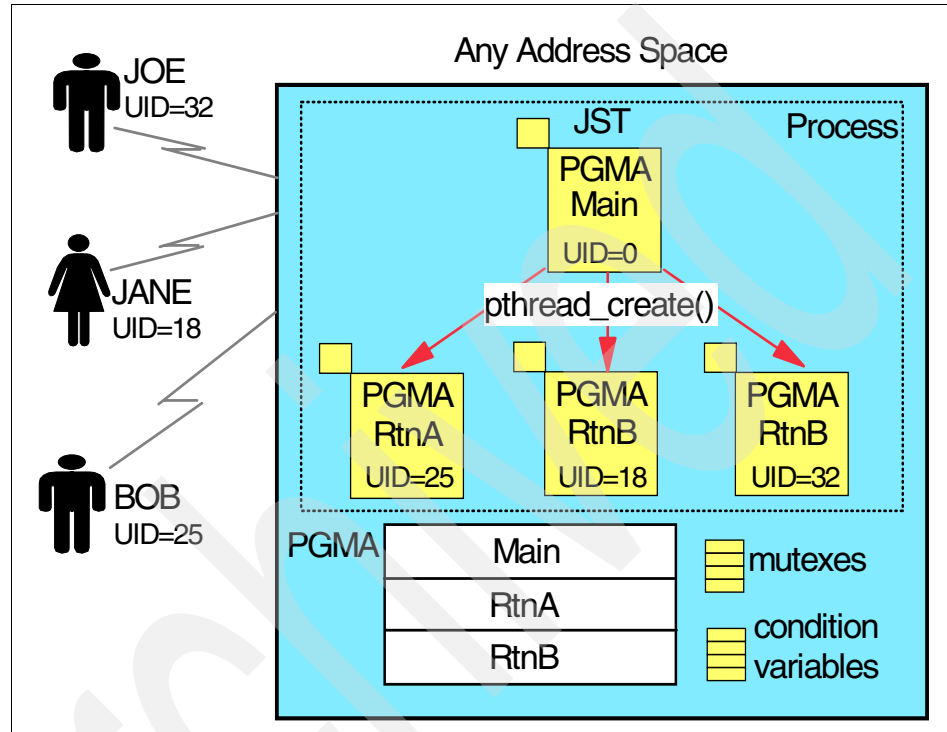


Figure 4-2 The UNIX thread concept and the z/OS UNIX implementation

4.2 Identities associated with a z/OS UNIX process or thread

Globally speaking, a z/OS task, after it has been dubbed as a UNIX process, is associated with two sets of identity: A set of UIDs and a set of GIDs.

Both sets, to conform with the POSIX.1 security model, actually contain three forms of the identity information: The *real* identity, the *effective* identity, and the *saved* identity.

4.2.1 Real and effective UID and GID

The effective UID is the UID value that the system uses for access control decisions. The real UID is actually the user UID under which the process has been initiated with. At process creation, the effective UID is made equal to the real UID. Likewise, the effective GID is used to determine the user's permission related to group access to files and directories. It can be different from the real GID, which is the GID of the group that the user is a member of when logging in to the system.

The real and effective IDs are generally the same for a process unless a `set-uid` or `set-gid` program is executed or functions that switch identity are called. The switched-to identity becomes the effective UID, and similarly the effective GID is changed to reflect the new effective UID group membership.

4.2.2 The saved UID and saved GID

These are entities that have been implemented to keep track of the UID or GID values that were in use before switching identities. When done with the switched-to identity, a program can use the `setuid()` or `setgid()` function without specifying a target identity to restore the saved UID or GID as the real or effective UID or GID.

4.3 Functions that change the effective UID and GID

The following functions, or their equivalent assembler calls, can change the effective UID and GID:

- ▶ `Setuid()`
- ▶ `Seteuid()`
- ▶ `Setreuid()`
- ▶ `Spawn()` with identity change
- ▶ `Pthread_security_np()`
- ▶ `_check_resource_auth_np()`
- ▶ `_login()`
- ▶ `_password()`
- ▶ `su`
- ▶ `setUID` flag

Identity switching is discussed in further detail in Chapter 5, “The z/OS UNIX security model” on page 51.

Archived

The z/OS UNIX security model

This chapter describes the implementation in z/OS UNIX of the superuser concept, and how the z/OS UNIX superuser privileges can be restricted by the RACF administrator.

From the principle standpoint, a superuser can do anything in a UNIX environment. In z/OS UNIX, this is also true as far as UNIX objects are considered. However, having UID(0) in z/OS UNIX does not provide any privileges over MVS objects or functions. For instance, the z/OS UNIX user, with the superuser status, does not acquire any rights on MVS data sets nor does he or she get any RACF SPECIAL privilege.

But still only very trusted persons should be deemed the superuser status as any misuse of their privileges can have quite detrimental consequences to both the z/OS UNIX users and the environment.

5.1 The superuser concept and privileges

The superuser is one of the UNIX basic concepts and as such must be supported by z/OS UNIX. A typical example for a requirement to have superuser authority is a UNIX server daemon, which must be able to read or update any UNIX file on behalf of any user. Such applications might have to be ported onto z/OS UNIX, and actually some are.

Furthermore, for system-wide tasks such as managing files and file systems, you need a user who can:

- ▶ Create a new user's home directory and make the new user owner of this directory
- ▶ Delete a user's home directory with all its content
- ▶ Mount and unmount other file systems at selected mount points
- ▶ Quiesce the file system

Superusers are the UNIX answer to these requirements. Superusers are originally intended to pass all security checks for UNIX resources access. They can change their identity to any UID and can increase their own limits for UNIX resources consumption as opposed to the limits assigned to the rest of the system.

5.1.1 The concerns with the superuser concept

One drawback of the superuser concept is its very ability to do anything at any time in the system. It is sometimes difficult to recover from their improper actions such as misspellings in commands.

The concept also does not fit well in environments with stringent security where duties and privileges tend to be carefully separated among users and assigned in a granular way. In addition, this situation is even aggravated today with the many regulations that banking or health organizations have to comply with.

As a parallel, the OPERATIONS attribute that can be given in RACF to some MVS users is a sort of limited superuser. Users with this attribute have full access to files, provided they are not already explicitly designated on the resource profile access list, in which case the access they are given in the access list takes precedence over the OPERATIONS attribute. They have the same limited *super access* to some general resource classes such as TAPEVOL and DASDVOL.

z/OS installations have been reducing, for years, the number of user IDs with the OPERATIONS attribute, to a point where it is almost not used anymore. For instance COPY MERGECAT operation in IDCAMS is one of the few valid justifications for OPERATIONS. Likewise, z/OS UNIX proposes ways of reducing the amount of users with UID(0), as explained in 5.6, “z/OS UNIX users privilege granularity” on page 64.

5.2 z/OS UNIX implementation of the superuser concept and privileges

As already mentioned, z/OS UNIX supports the superuser concept for users who are assigned a UID(0). This UID can be assigned in the OMVS segment of the USER profile, or could (*and there is a strong advice not to do so*) be assigned via the BPX.DEFAULT.USER profile in the FACILITY class.

However, z/OS UNIX implements optional tighter controls on what a superuser can do, which are explained in this chapter.

An important superuser privilege is the capability of switching the identity of its executing process. This capability does exist in the z/OS UNIX implementation of the superuser concept and leads to specific considerations because of the z/OS UNIX dual user identity. That is the correspondence that is established between the MVS user ID and the UID. These considerations are developed in the following section.

Note: Started tasks running with the TRUSTED or PRIVILEGED attribute are considered z/OS UNIX superusers when they are dubbed even if their assigned UID is a value other than 0.

5.2.1 Reminder on z/OS UNIX identity switching

When a job starts or a user logs on to an application, the MVS user ID and password are verified by invoking existing z/OS and RACF functions. We mentioned that when the address space requests a z/OS UNIX function for the first time, the address space is dubbed, meaning that it is now considered as a UNIX process. The RACF involvement in the dubbing process initialized by the OMVS kernel is as follows:

- ▶ It verifies that the user is defined as a z/OS UNIX user, via an OMVS segment or via the BPX.DEFAULT.USER profile.

- ▶ It verifies that the user's default group and current connect group (or any group the user is connected to if RACF list-of-groups is in effect) is defined as a z/OS UNIX group.
- ▶ It updates the control blocks needed for subsequent security checks, both with the MVS and the UNIX identities.

A very important point to remember here is that a z/OS UNIX program authorized to switch to an alternate UID value on z/OS will also switch in most cases to the corresponding MVS user ID, in order to always conform to the dual identity as specified in the RACF USER profile.

Great care must be exercised when dealing with these UNIX programs if they also access MVS resources or invoke MVS authorized services, as they are getting the privileges of the switched-to MVS user ID. It is important to understand if the switched-to identity is first authenticated, using the UNIX user name (actually for z/OS UNIX the MVS user ID, as it would be the case for MVS programs intended to switch task identity), or if the switching is done using superuser authority and therefore does not require authentication of the target identity.

The z/OS UNIX approach is to tightly control the superusers' authority to switch identity. These controls use RACF profiles that the superuser has to be permitted to, along with an extension of the RACF *Program Control* function.

z/OS UNIX also exploits the SURROGAT class of profiles in RACF, which can be used to authorize bypassing authentication when switching to a specific user ID.

Table 5-1 summarizes the conditions under which a z/OS UNIX program that switches its UID also switches its MVS user ID. "Y" indicates that an identity switch is performed.

Table 5-1 UNIX UID and MVS user ID changes

UNIX System Services functions that result in UID switching	Change effective UID	Change MVS user ID ^a	Conditions for switching identity Note: We are assuming that the RACF BPX.DAEMON profile has been defined in the FACILITY class ^b
setuid flag in the file security packet	Y	N	Only the effective UID is changed for the time of the program execution
Non-superuser issues su in shell, without specifying a user ID	Y switches to UID(0)	N	The MVS user ID needs READ access to BPX.SUPERUSER in the FACILITY class
su with user ID and valid password	Y	Y	A valid password for the target user ID is required

UNIX System Services functions that result in UID switching	Change effective UID	Change MVS user ID ^a	Conditions for switching identity Note: We are assuming that the RACF BPX.DAEMON profile has been defined in the FACILITY class ^b
su with user ID without password	Y	Y	The issuing MVS user ID needs READ access to the target user ID profile in the SURROGAT class
setuid() or _login() with authentication of the target ID	Y	Y	The function must be invoked from a <i>clean</i> address space ^c
setuid() or _login() without authentication of the target ID	Y	Y	The requesting MVS user ID needs READ access to the target user ID profile in the SURROGAT class OR The requesting MVS user ID has UID(0) AND has READ access to BPX.DAEMON AND the function is invoked from a clean address space (see Note c below)
pthread_security_np() with authentication of target ID	Y	Y	The requesting MVS user ID needs at least READ access to BPX.SERVER AND the function is invoked from a clean address space (see Note c below)
pthread_security_np() without authentication of target ID	Y	Y	The requesting MVS user ID needs READ access to the target user ID profile in the SURROGAT class AND The requesting MVS user ID has at least READ access to BPX.SERVER AND the function is invoked from a clean address space (see Note c below)
_spawn() with identity change with authentication of target ID	Y	Y	
_spawn() with identity change without authentication of target ID	Y	Y	The requesting MVS user ID needs READ access to the target user ID profile in the SURROGAT class OR The requesting MVS user ID has UID(0) AND has READ access to BPX.DAEMON AND the function is invoked from a clean address space (see Note c below)

a. When changing from a non-zero UID for UID(0), the MVS user ID is not changed.

b. When BPX.DAEMON is not defined, z/OS UNIX operates as any UNIX system, granting full uncontrolled authority to superusers.

Note: If the BPX.SERVER or BPX.DAEMON in the FACILITY class is defined, the system is said to have z/OS UNIX-level security. In this case, the system is more secure than a traditional UNIX system.

c. The notion of *clean* address space is explained in 5.3, “Introducing the controlled environment” on page 58.

5.2.2 Authentication of the switched-to user ID

z/OS UNIX offers a peculiar context when it comes to a UNIX application to switch identity as, because of the dual identity associated to the z/OS UNIX users, the MVS identity might have to be switched as well. In this section, we provide the operating rules followed by z/OS UNIX to associate a new MVS identity, if required to, to the switched-to UID.

Switching UID without authentication of the target UID

Some UNIX daemon processes might issue a `setuid()` to switch to any UID without authentication of the target user ID. When the BPX.DAEMON FACILITY profile is defined, the identity switching occurs if the caller has UID(0) and is permitted to the BPX.DAEMON profile (see below).

The switched-to MVS user ID is a user who has the same UID as the target UID. This is another reason to establish unique UIDs, although it might not be possible if the target UID is UID(0).

The `setuid()` function invokes the System Authorization Facility (SAF) services to change the MVS identity of the address space. The MVS identity that is used is determined as follows:

- ▶ If an MVS user ID is already known by the kernel from a previous call to a kernel function (for example, `getpwnam()`) and the UID for this user ID matches the UID specified on the `setuid()` call, then this user ID is used.
- ▶ For non-zero target UIDs, if there is no saved user ID or the UID for the saved user ID does not match the UID requested on the `setuid()` call, the `setuid()` function queries the security database (for example, using the `getpwnam()` function) to retrieve a user ID that also has the target UID. The retrieved user ID is then used.

Switching to UID(0) without authentication

There are cases in UNIX programming where the superuser might want to switch to UID(0) (actually this is a way to check if a program is already running with UID(0)). z/OS UNIX uses a default target user ID in the case of BPXROOT (that is, a BPXROOT USER profile, with UID(0) is expected to have been defined in

RACF). A different default user ID can be specified with the SUPERUSER keyword in the SYS1.PARMLIB(BPXPRMxx) member.

Tips:

- ▶ It is recommended that the UID(0) default user (BPXROOT or other) be defined with the NOPASSWORD attribute.
- ▶ An MVS user defined with an OMVS segment with NOUID will never be switched to with `setuid()`.

Important: Remember that if BPX.DEFAULT.USER is defined, then any user without an OMVS segment can be the subject of an identity switch in a UNIX process or thread.

5.2.3 The RACF BPX.DAEMON profile in the FACILITY class

If the BPX.DAEMON resource in the FACILITY class is defined, the system is said to have “z/OS UNIX security”. Implying that it has a superior level of user control than initially defined in the UNIX security model.

In regular UNIX systems, superusers can freely change their identity for another one at any time without justifying of any permission to do so. As a parallel in the MVS world, a task can change identity only if it can provide the password of the target identity or if the caller has been defined by the RACF administrator as a *surrogate* of the target identity. Non-superusers in UNIX can also change their identity only if they can provide the valid password for the target identity.

With the BPX.DAEMON defined in the FACILITY class, a process running under a superuser identity can freely switch, that is without having to provide the target identity password, to any new identity if the both following statements are true:

- ▶ The caller’s MVS user ID identity has READ access to the BPX.DAEMON profile.
- ▶ The request is issued from a *clean* address space, also called a *controlled environment*. We explain in further detail what a controlled environment is in 5.3, “Introducing the controlled environment” on page 58.

Important:

- ▶ It is highly recommended to have the BPX.DAEMON profile defined in the FACILITY class, with UACC(NONE). The profile's access list is then updated on a case-by-case basis.
- ▶ The default superuser user ID BPXROOT (or any alternate user ID defined in SYS1.PARMLIB(PRMXX)) must not be permitted to BPX.DAEMON.
- ▶ After BPX.DAEMON has been defined, the system should be set up to establish a controlled environment such as explained in 5.3, "Introducing the controlled environment" below.

5.3 Introducing the controlled environment

The controlled environment is the exploitation by z/OS UNIX of the program control mechanisms. Program control was implemented many years ago in MVS so that the system can control the accesses that users have to load modules and control the accesses that the programs themselves have on data sets. This control is achieved by defining resource profiles in the RACF PROGRAM class of profiles to specify which load modules, or libraries, are program controlled.

Having the RACF administrator defining a program in the PROGRAM class is in itself an indication that some level of trust is granted to the program contents, as it could be the case after inspecting the program's code for sound and safe design. Programs that can be controlled in this way include programs loaded from the LNKLST (called public libraries) or from JOBLIB/STEPLIB (private libraries). CLISTs, procedures, and LPA modules cannot be controlled.

The program control environment is activated with the RACF command:

```
SETOPTS WHEN(PROGRAM)
```

Program control goes along with additional processing that checks, when loading a load module into an address space, that the load module is program controlled.

This notion of controlled program has been extended to UNIX executable modules residing in Hierarchical File System (HFS) or zFS files. As HFS or zFS files are not resources defined in RACF, the controlled program indication is kept in the file security packet as the "p" extended attribute.

A clean address space contains only modules that are either MVS controlled programs (that is defined in the PROGRAM class) or UNIX executables loaded from files with the “p” extended attribute. Conversely an environment on which a program not defined in the PROGRAM class, or fetched from a file without the “p” extended attribute has been loaded, is considered a *dirty* or *uncontrolled environment*.

A schematic view of the mechanism is given in Figure 5-1 on page 60. As programs are loaded from MVS library or from UNIX files, the program load function checks whether the load module is defined as a PROGRAM resource in RACF (for MVS load modules) or whether the file has the “p” extended attribute (for UNIX executable files). If this is not the case, the program is still loaded in the requestor’s address space, however an indication, known as the *dirty bit*, is set in the address space control blocks.

The following functions that can be invoked by a program running in the address space and that eventually, for some of them, proceed with a switch of identity, require, in order to be executed, that the issuing address space be a clean address space. If this is not the case, the function is not executed and control is given back to the requesting program with an error information.

- ▶ Setuid()
- ▶ Seteuid()
- ▶ Setreuid()
- ▶ Spawn() with identity change
- ▶ Pthread_security_np()
- ▶ _check_resource_auth_np()
- ▶ _login()
- ▶ _password()
- ▶ su
- ▶ setUID flag

The implementation of these function calls in z/OS UNIX is as shown in Figure 5-1 on page 60. Calling the function gives control to the z/OS UNIX kernel and the kernel checks whether the BPX.DAEMON profile has been defined and whether the dirty bit is on or not for the requestor’s address space.

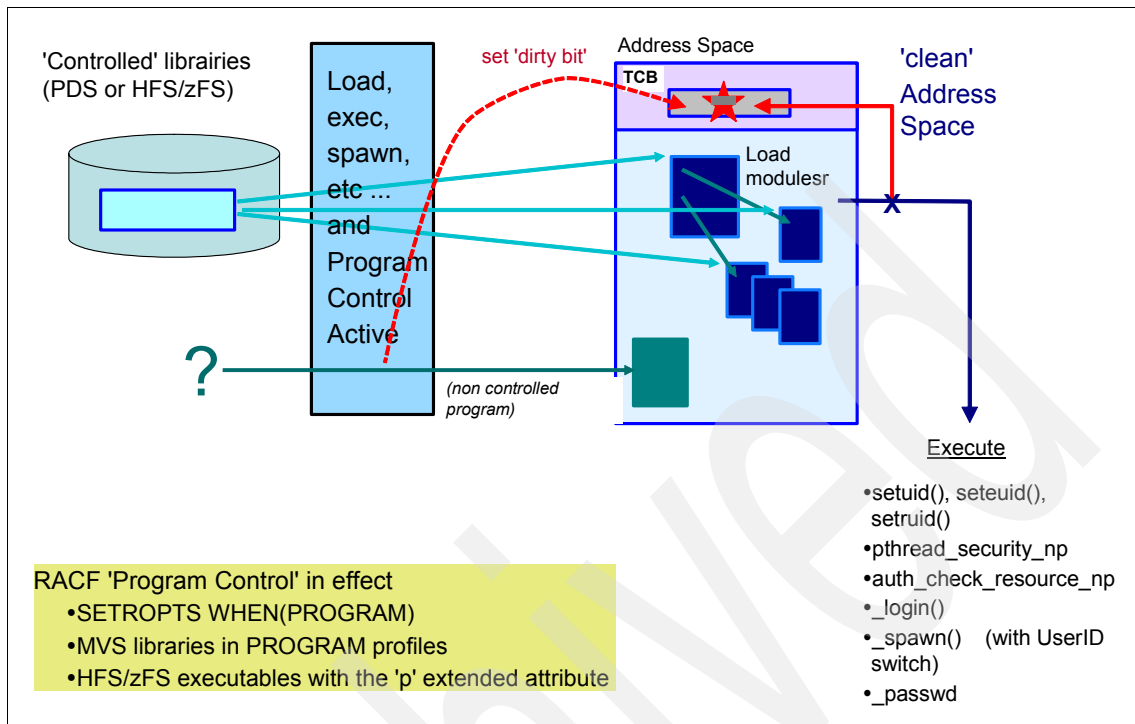


Figure 5-1 The controlled environment process

Important: There is no program control checking for program modules loaded from the LPA, that is, they are considered to be controlled.

What it takes to establish a controlled environment

Here is an example command to get a program to be program controlled:

```
RDEFINE PROGRAM ABC ADDMEM('SYS1.LINKLIB'//NOPADCHK) UACC(READ)
SETROPTS WHEN(PROGRAM)
```

In this example, the program "ABC" in the SYS1.LINKLIB library is controlled. Note that giving READ access by default does not provide any restriction to its use by non-restricted RACF users, the purpose being here to just get this program's execution environment to be controlled as explained above. A variation of this setup is to get a complete program library to be controlled:

```
RALTER PROGRAM * ADDMEM('SYS1.SGSKLOAD'//NOPADCHK) UACC(READ)
SETROPTS WHEN(PROGRAM) REFRESH
```

This last example can be used to make all the z/OS LINKLIST libraries controlled as members of the PROGRAM * profile. Be careful that with new releases of z/OS new libraries may be on the LINKLIST, and the PROGRAM * profile in RACF may need to be updated accordingly.

Executable programs that reside in file system files are marked with the “p” extended attribute bit on to indicate that they are program controlled. Note that in that case, each program has to be individually marked as controlled as the file system cannot be considered as a controlled library.

Important: To set the UNIX file “p” extended attribute requires having, in addition to being the file owner or a superuser, READ permission to the BPX.FILEATTR.PROGCTL profile in the RACF FACILITY class. Note also that any change to the file contents automatically resets the “p” bit.

BPX.DAEMON.HFSCTL

The capability exists for selected applications not to check whether program loaded from MVS libraries are program controlled, but to control only UNIX executables for coming from files with the “p” extended attribute. In order to do so, the profile BPX.DAEMON.HFSCTL has to be defined in the FACILITY class and the applications user IDs for whom the MVS program control check has to be bypassed are given READ permission to the profile.

Obviously, doing this weakens some of the security provided by the BPX.DAEMON resource. It should be done only in restricted and carefully considered cases, or if you do not already run with BPX.DAEMON but want to gain only a subset of the benefits of running with BPX.DAEMON.

BPX.DAEMON.HFSCTL is an example of a profile, that potentially *lowers* the ambitions of a system security policy.

Recommendation: You must not define a generic FACILITY catchall profile such as ** even if it has UACC(NONE), because it will cause by virtue of its very definition policy setting entities such as BPX.DAEMON.HFSCTL to exist.

5.4 Using surrogate users with z/OS UNIX

Surrogate users is a facility in the MVS RACF security model that can be exploited in a z/OS UNIX environment. A surrogate user can act on behalf of another user, under control of RACF administrative setup and gets the access privileges of the other users without having to provide the surrogated user’s authentication data.

In order to make a z/OS UNIX user the surrogate of another z/OS UNIX user, the RACF administrator has to define the profile `BPX.SRV.surrogated_id` in the `SURROGAT` class, the last qualifier in the profile name being the user ID of the surrogated user. Then users of the RACF user ID that has `READ` access to the profile are permitted to act as a surrogate of *surrogated_id*.

We have already indicated in the previous sections that switching user identity is allowed without further control if the switching user ID is defined as a surrogate of the switched-to user ID.

5.5 BPX.SERVER

The `BPX.SERVER` is a RACF profile in the `FACILITY` class that is used to protect access to the functions `pthread_security_np()` and `_check_resource_auth_np()`.

`pthread_security_np()` is invoked by server processes that create individual threads for the requests they serve and have to explicitly assign an identity to these threads of execution, that is, they have to establish a specific security environment for the subtask to be run in the address space. It is expected that such server processes first authenticate the client and then create the thread security environment using `pthread_security_np()` with the client authentication data. If they do not proceed with client authentication, as it could be the case with a default user ID or an anonymous request, then the server's address space user ID has to be defined as a surrogate of the default or anonymous user ID.

When the `BPX.SERVER` profile is not defined, only superusers can invoke the `pthread_security_np()` and `_check_resource_auth_np()` functions. However, when the profile is defined only users having at least `READ` permission to the profile, be it superuser or not, can use the `pthread_security_np()` functions, as well as the `_check_resource_auth_np()` function. Execution of these two functions requires the issuing address space to be clean. Figure 5-2 on page 63 illustrates the creation of a thread security context with the `pthread_security_np()` function.

It is also assumed that the server process user ID has been defined as a RACF surrogate for the client.

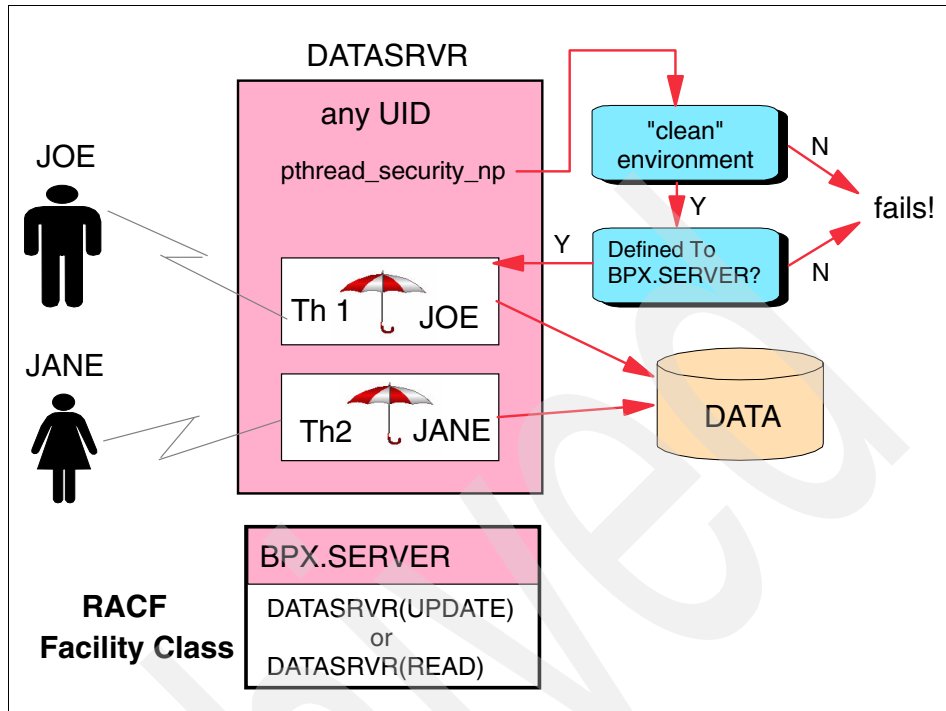


Figure 5-2 *pthread_security_np* and *BPX.SERVER*

Note that there is a difference whether a server process is given READ or UPDATE permission to the *BPX.SERVER*.

UPDATE: When the RACF identity of the server has been granted UPDATE authority to *BPX.SERVER* in the RACF FACILITY class, the server fully acts as a surrogate for the client. This means that the identity of the thread associated with the request from the server's client runs with the z/OS user ID of the server's client. Access control decisions to z/OS resources (such as data sets) and to z/OS UNIX resources (such as UNIX files) that are accessed by the client's thread in the server are made using the RACF identity of the client.

READ: READ access lets the server establish a thread-level security environment for the clients that it services. However, the user ID of the server and the user ID of the client must be authorized to the resources that the server will access. A thread-level security context in which both the client's and server's identity is used in the access control decision and a password was not supplied by the client is called an unauthenticated client security context. Depending on the design and implementation of the client/server application, a client may have to supply an authenticator to the server. For example, the client may be

prompted to supply a password or a password substitute, such as a RACF passticket to the server to prove its identity.

If a RACF password or passticket is specified as a parameter on the `pthread_security_np()` service, and the password or passticket is valid for the client user ID, even if the server's identity has been granted READ access to the profile `BPX.SERVER` in the RACF FACILITY class, the task level security environment is only used in access control decisions. That is, only the RACF user ID of the client is used in making access control decisions. This task level security environment created by a server is called an authenticated client security context. Because the client has trusted the server sufficiently to supply a RACF password (or passticket) to the server, the server is granted the capability of acting as a surrogate for that client (user).

Note: It is also recommended to have `BPX.SERVER` defined with `UACC(NONE)`.

5.6 z/OS UNIX users privilege granularity

z/OS UNIX offers ways of maintaining the amount of UNIX users defined with UID(0) to a bare minimum. The approach is to grant superuser privileges on a temporary basis, or to grant granular pieces of superuser privileges to regular UNIX users (that is non-UID(0) users) under control of RACF profiles.

5.6.1 BPX.SUPERUSER

The `BPX.SUPERUSER` profile can be defined in the FACILITY class. Regular UNIX users permitted to this resource can switch to superuser status by issuing the `su` ("switch user") shell command in the z/OS UNIX shell. This initiates a nested shell where the user has superuser status. The user keeps the superuser status until exiting from this nested shell.

The objective is that users permitted to `BPX.SUPERUSER` can temporarily acquire superuser privileges when, and only when, they explicitly want to exploit them. Note that these users can also switch into the superuser status with the "Enable superuser mode (SU)" option in the ISPF shell, and remain with this status until they explicitly select the option to reset to their original UID's privileges.

5.7 Individual limits in the USER profiles

The BPXPRMxx member in SYS1.PARMLIB is used to specify limits pertaining to resources allocation intended for all z/OS UNIX users. For instance, the parameter MAXTHREADTASKS is the maximum number of MVS tasks that a single process can have concurrently active.

Processes that need, for any reason, to go above the general limits fixed in the BPXPRMxx member can either:

- ▶ Be assigned a z/OS UNIX user ID with a UID(0). Superusers are not subject to the general limits.
- ▶ Or run with a non-UID(0) user ID with individual limits specified in the USER profile. The individual limits override the general limits given in BPXPRMxx.

These individual limits are specified in the OMVS segment of the USER profile with the following keywords:

- ▶ ASSIZEMAX(address-space-size)
- ▶ CPUTIMEMAX(cpu-time)]
- ▶ FILEPROCMAx(files-per-process)
- ▶ MEMLIMIT(nonshared-memory-size) | NOMEMLIMIT
- ▶ MMAPAREAMAX(memory-map-size)
- ▶ PROCUSERMAX(processes-per-UID)
- ▶ SHMEMMAX(shared-memory-size) | NOSHMEMMAX
- ▶ THREADSMAX(threads-per-process)

Note that these individual limits, although expected to be mainly used for increasing the consumption limits for some resources, can also be used to decrease the user's limits with respect to the general limits.

To use the `_BPX_UNLIMITED_SPOOL` environment variable, the caller must be a superuser or be permitted to the BPX.UNLIMITED.SPOOL FACILITY class profile with READ access or greater. A "YES" value given to the environment variable specifies unlimited spooled output. "NO" specifies that the default spooled output limits is to be used. Not defining or specifying an invalid value is the equivalent of specifying NO and the defaults limits are not overridden.

5.7.1 The UNIXPRIV class of resources

By using the UNIXPRIV class, the RACF administrator can permit a subset of superuser privileges to non-UID(0) users. A number of system programmers and administrators do not need full superuser authority. They just need to perform a few selected functions that cannot normally be executed without superuser authority. In this situation, the number of superusers, or users able to switch into

superuser mode through access to BPX.SUPERUSER, that are needed in the system can be reduced by using *superuser granularity* via profiles in the RACF UNIXPRIV class.

Superuser granularity allows a non-superuser to successfully execute a function that normally requires superuser authority if the user has access to a certain resource in the UNIXPRIV class of profiles. For example, a file system can normally be mounted only by a superuser. However, a user with a non-zero UID can successfully mount file systems if the user is permitted to SUPERUSER.FILESYS.MOUNT profile. READ access will allow the user to mount file systems with the nosetuid option, while UPDATE access will allow the user also to mount file systems with the setuid option (see Chapter 6, “z/OS UNIX files security” on page 73).

Table 5-2 shows the resource names that are used in the UNIXPRIV class of profiles and lists the privileges associated with each resource. Note that profiles with an access required of NONE are profiles acting as *switches* to enable the function.

The UNIXPRIV class must be active and SETROPTS RACLIST must be in effect for the UNIXPRIV class (this implies that changes to the UNIXPRIV profiles must be followed by a SETR RACLIST(UNIXPRIV) REFRESH). Global access checking is not used for authorization checking to UNIXPRIV resources.

Note: In this section, we do not list all the profiles in the UNIXPRIV class; we list only the ones that directly pertain to superuser granular privileges granted to regular users. For a complete description of all profiles in the UNIXPRIV class, refer to *z/OS V1R8.0 UNIX System Services Planning*, GA22-7800.

Table 5-2 Resource names in the UNIXPRIV class for z/OS UNIX privileges

Resource name	z/OS UNIX privilege	Access required
CHOWN.UNRESTRICTED	Allows all users to use the chown command to transfer ownership of their own files.	NONE
SUPERUSER.FILESYS.CHOWN	Allows users to use the chown command to change ownership of any file.	READ

Resource name	z/OS UNIX privilege	Access required
SUPERUSER.FILESYS ^a	Allows users to read any HFS file and to read or search any HFS directory.	READ
	Allows users to write to any HFS file and includes privileges of READ access.	UPDATE
	Allows users to write to any HFS directory and includes privileges of UPDATE access.	CONTROL (or higher)
SUPERUSER.FILESYS.CHANGEPERMS	Allows users to use the chmod command to change the permission bits of any file and to use the setfac1 command to manage access control lists for any file.	READ
SUPERUSER.FILESYS.MOUNT	Allows users to issue the mount command with the nosetuid option and to unmount a file system mounted with the nosetuid option.	READ
	Allows users to issue the mount command with the setuid option and to unmount a file system mounted with the setuid option.	UPDATE
SUPERUSER.FILESYS.QUIESCE	Allows users to issue the quiesce and unquiesce commands for a file system mounted with the nosetuid option.	READ
	Allows users to issue the quiesce and unquiesce commands for a file system mounted with the setuid option.	UPDATE
SUPERUSER.FILESYS.PFSCTL	Allows users to use the pfsc1() callable service.	READ
SUPERUSER.FILESYS.VREGISTER ^b	Allows a server to use the vreg() callable service to register as a VFS file server.	READ
SUPERUSER.IPC.RMID	Allows users to issue the ipcrm command to release IPC resources.	READ
SUPERUSER.PROCESS.GETPSENT	Allows users to use the w_getpsent callable service to receive data for any process.	READ
SUPERUSER.PROCESS.KILL	Allows users to use the kill() callable service to send signals to any process.	READ

Resource name	z/OS UNIX privilege	Access required
SUPERUSER.PROCESS.PTRACE ^c	<ul style="list-style-type: none"> ▶ Allows users to use the <code>ptrace()</code> function through the <code>dbx</code> debugger to trace any process. ▶ Allows users of the <code>ps</code> command to output information about all processes. This is the default behavior of <code>ps</code> on most UNIX platforms. 	READ
SUPERUSER.SETPRIORITY	Allows users to increase their own priority.	READ

a. Authorization to the SUPERUSER.FILESYS resource provides privileges to access only local Hierarchical File System files. No authorization to access Network File System (NFS) files is provided by access to this resource.

b. The SUPERUSER.FILESYS.VREGISTER resource authorizes only servers, such as NFS servers, to register as file servers. Users who connect as clients through file server systems, such as NFS, are not authorized through this resource.

c. Authorization to the resource BPX.DEBUG in the FACILITY class is also required to trace processes that run with APF authority or BPX.SERVER authority. For more information about administering BPX profiles, see *z/OS V1R8.0 UNIX System Services Planning, GA22-7800*.

5.8 Some recommendations

We found the following RACF setups to highly contribute to the z/OS UNIX environment security. Their use depends of course on the specific installation context, but we believe that they should be considered as part of the recommended *best practices* related to the z/OS UNIX security setup.

- ▶ SETROPTS NOADDCREATOR
- ▶ RDEFINE FACILITY BPX.DAEMON UACC(NONE) OWNER(SYS1)
- ▶ PERMIT BPX.DAEMON CLASS(FACILITY) ID(BPXROOT) ACCESS(NONE)
- ▶ RDEFINE FACILITY BPX.SERVER UACC(NONE) OWNER(SYS1)
- ▶ RDEFINE SURROGAT BPX.SRV.* OWNER(SYS1)

We also stress the usefulness of funneling UID(0) requests into a limited set of administrators with READ access to SHARED.IDS. Furthermore, if a user is requesting UID(0), he or she will have to justify it to a senior administrator who would rather look into the use of UNIXPRIV profiles as a replacement for UID(0).

5.9 Other restrictions to superuser authority

Other restrictions to superuser authority include:

- ▶ Extended attributes for HFS files such as program-controlled “p” bit, APF-authorized “a” bit, and shared library “l” bit can only be set with READ authority to the appropriate RACF profile in the FACILITY class: BPX.FILEATTR.PROGCTL, BPX.FILEATTR.APF, or BPX.FILEATTR.SHARELIB, respectively. See Chapter 8, “Considerations on z/OS UNIX program management” on page 105, for further details on these bits.
- ▶ Use of the ptrace function of dbx to debug programs running with APF authority or with BPX.SERVER authority requires READ access to the profile BPX.DEBUG in the FACILITY class.

5.10 The daemons in z/OS

As of z/OS V1R7, the following UNIX programs are delivered in z/OS and operate with superuser authority to switch to unauthenticated identities. They have been thoroughly reviewed by IBM so that they can be permitted to the BPX.DAEMON FACILITY profile without introducing a security exposure.

- ▶ inetd: The network daemon
- ▶ rlogind: The remote login daemon
- ▶ cron: The clock daemon
- ▶ uucpd: The UUCP daemon
- ▶ The syslogd daemon

5.11 Advanced topic: RACF enhanced program security

In this section, we provide an overview of this enhancement to the program control protection mechanism. Further details pertaining to its principles of operation and administration can be found in *z/OS V1R8.0 Security Server RACF Security Administrator's Guide*, SA22-7683, and *z/OS V1R8.0 UNIX System Services Planning*, GA22-7800.

5.11.1 Overview of the principles of operation

We have seen in 5.3, “Introducing the controlled environment” on page 58, how the MVS program control mechanism is exploited in z/OS UNIX to check that critical functions, mainly dealing with identity switching, require, in addition to

being authorized to the BPX.DAEMON or BPX.SERVER profiles, to be called from a clean address space.

RACF program control, control of program access to data sets (PADS), and protection of programs with sensitive data or/and algorithm inside (EXECUTE control) have been enhanced at z/OS V1R4 with the introduction of RACF enhanced program security.

RACF enhanced program security provides a finer level of checking for controlled programs that makes it even more difficult for malicious users to establish an illegitimate clean environment.

These optional controlled programs characteristics have to be entered in the APPLDATA field of the PROGRAM profiles. They are:

- ▶ “MAIN”: Programs with MAIN in the APPLDATA field of their PROGRAM profile are tracked to be the initiator of the environment. If the environment is initiated by a non-MAIN program, then the environment becomes dirty. Typically MAIN programs are programs started by:
 - // EXEC PGM=program
 - TSOEXEC program
- ▶ “BASIC”: Programs with the BASIC attribute are the first program of the current task or a parent task. The check for clean environment only involves the library or file or data set the program is coming from.

Here are examples of definition for a MAIN program and BASIC program:

```
RDEFINE PROGRAM ABC ADDMEM('load.library'//NOPADCHK) APPLDATA('MAIN')
RDEFINE PROGRAM ABC ADDMEM('load.library'//NOPADCHK) APPLDATA('BASIC')
```

In the above example, you can declare the program ABC as MAIN, if it is called via a JCL EXEC. However, if you start the program ABC with a TSO CALL or ISPEXEC, ABC does not create the environment (IKJEFT01 did it) and therefore can only be defined as BASIC. If the MAIN program resides in the shared storage area, LPA, use 'LPALST' as library name, and in this case the UACC value of the PROGRAM profile is not being used.

These programs characteristics are honored when the system is running in enhanced program mode. To turn on enhanced program mode in the system, you have to define the mode in the APPLDATA field of the profile IRR.PGMSECURITY in the RACF FACILITY class. The mode is one of the following:

- ▶ “BASIC” mode: This is the pre-enhanced program security mode, or the default mode. It only checks for proper program controlled origin of the modules brought into the address space.

- ▶ “ENHANCED” mode: The system is to ensure that MAIN programs are really the initiator of the environment. BASIC programs are checked, as done already, for a program controlled source.
- ▶ “ENHANCED-WARNING” mode: This is an aid for the migration to ENHANCED mode, as running with ENHANCED mode is probable to fail the first times as many programs that might be found not to be properly controlled yet.

Important: Switching mode in the IRR.PGMSECURITY profile requires to re-IPL the system for all tasks to execute in the selected mode.

5.11.2 Enhanced program security and z/OS UNIX

It appears from the explanations above that the MAIN or BASIC attribute can only be specified in PROGRAM profiles in RACF, that is there is no MAIN or BASIC program characteristics that can be set for the z/OS UNIX executable files.

However, you can define the FACILITY profile BPX.MAINCHECK. With this profile defined, z/OS UNIX and RACF will require that the first program a daemon executes must be defined to RACF using a PROGRAM profile with the MAIN option, as described previously. If you define BPX.MAINCHECK, then the first program that any daemon executes has to be moved first into an MVS library and defined in the RACF PROGRAM profiles with the MAIN attribute. The initial z/OS UNIX executable needs to have the sticky bit attribute turned on or can be set up as an external link z/OS UNIX file.

Remember that programs loaded from the LPA are all considered to be program controlled.

5.12 A word on IPC security

Interprocess communications (IPC) are, with files and directories, part of the UNIX applications resources that are protected. In z/OS UNIX, IPC comprises the following facilities, according to the XPG4 support:

- ▶ **Message queues:** Message queues allow a client and a server process to communicate through one or more message queues in the kernel. A process can create, read from, or write to a message queue. Multiple client and server processes can share the same queue.
- ▶ **Shared memory:** Shared memory provides a method of sharing data in storage between multiple processes. The shared data is kept in a data space

created by the kernel. The data can be shared between a parent and child process or between unrelated processes.

- ▶ **Semaphores:** Semaphores are used for serializing access to shared memory. A program using shared memory must get a semaphore before it allocates shared memory.

Note: The IPC mechanisms have a limited set of permission bits (read and write for user, group, and other), which can be manipulated by APIs, but not by commands.

IPC requires RACF to do authorization and permission checking. IPC facilities of the z/OS UNIX system allow two or more distinct processes to communicate with each other. RACF protects this environment so that only those processes with the correct authority can communicate. IPC consists of message queuing, semaphores, and shared memory segments used by application programs. Each function requires a security action by z/OS UNIX, which RACF performs to allow a secure environment to exist.

The IPC security packet (ISP) contains data needed to make security decisions. It is built when a new ID for an IPC key is created and is saved in memory by the kernel. An IPC key is a number used to identify unequivocally an IPC control structure and is usually created with the `ftok(3)` function. The ISP is used in place of a profile in the RACF database to contain information about the IPC key's owner and access rights.

Access READ to the profile SUPERUSER.IPC.RMID in the class UNIXPRIV allows users to issue the `ipcrm` command to release IPC resources.

z/OS UNIX files security

In this chapter, we address the security of the MVS data sets that host the z/OS UNIX users' data and how the security model of the UNIX Hierarchical File System (HFS) is implemented in z/OS UNIX.

In this chapter, we address:

- ▶ The HFS and the file security packet (FSP)
- ▶ The pertinent z/OS UNIX security checks

6.1 z/OS implementation of the Hierarchical File System

As seen in Chapter 1, “Overview of the UNIX operating system model” on page 1, UNIX provides a Hierarchical File System, made of directories and files within the directories for the UNIX users and operating system to store and retrieve data. This section gives an overview of the UNIX Hierarchical File System implementation as done in z/OS with a specific focus on the protection of both the data sets that contain the z/OS UNIX data and the directories and files as they appear to the z/OS UNIX users.

6.1.1 The z/OS UNIX file systems

The file system implementations that we focus on in this chapter are the z/OS HFS and z File System (zFS). Other file systems are also supported in z/OS, such as Network File System (NFS) and Temporary File System (TFS). Although they abide with the basic security model that we describe for HFS and zFS, they do not directly involve z/OS data sets and require specific setups that we do not address in this IBM Redpaper. Further information about NFS can be found in *z/OS V1R8.0 Network File System Guide and Reference*, SC26-7417.

The z/OS UNIX file systems data is actually kept in data sets managed by DFSMS with a specific type of HFS or VSAM linear (the latter for the zFS file system). The z/OS UNIX logical file system gives the UNIX users the view of the data residing in the HFS or zFS data sets as though they were in files and directories.

In z/OS terminology, a *file system* is one data set that contains z/OS UNIX data. A z/OS system usually has many file systems that make up the full UNIX file tree. One data set hosts the *root* file system and contains the top of the files and directories hierarchy, while the lower levels of the hierarchy are stored in other file systems, that is data sets, that provide the lower levels of the hierarchy by being logically *mounted* at a directory of the next higher level, as shown in Figure 6-1 on page 75.

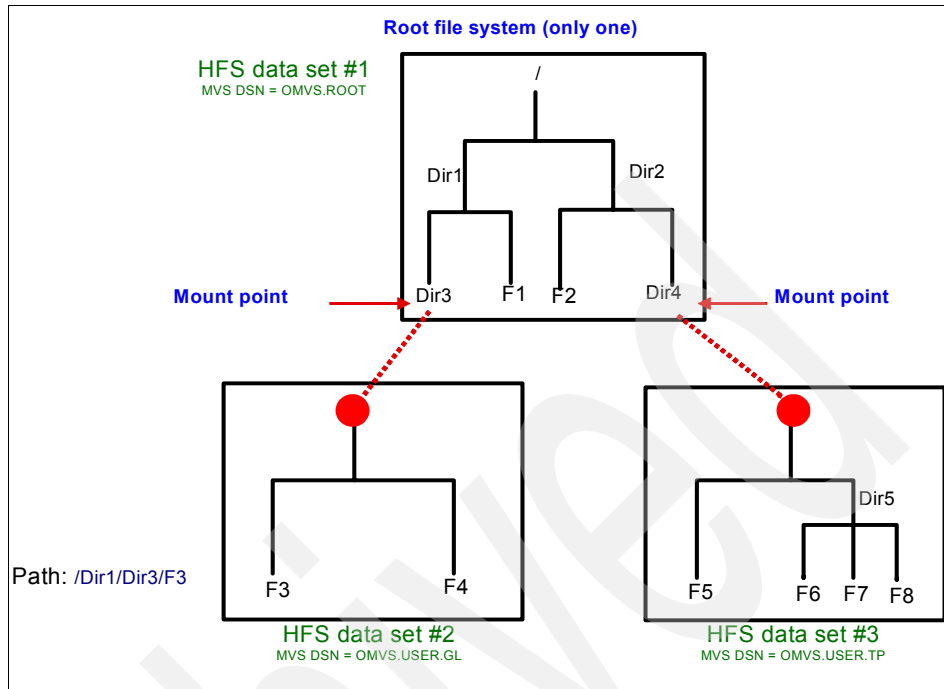


Figure 6-1 The z/OS UNIX Hierarchical File System

A file system is mounted with the TSO MOUNT, or UNIX mount, command which specifies the directory that the file system has to be mounted under. SYS1.PARMLIB(BPXPRMxx) contains the MOUNT commands to be automatically issued at z/OS UNIX initialization.

6.1.2 Protection of the file system data sets

These data sets must be defined in the RACF DATASET class with UACC(NONE). The z/OS UNIX kernel address space must have access to these data sets. The kernel initialization OMVS STC being defined with the TRUSTED attribute, or its user ID is given UPDATE permission to the Hierarchical File System data sets.

A very common convention is to give these data sets a high-level qualifier of OMVS.

Recommendation: It is recommended not to give the user's name for the high-level qualifier for the file system data set, because doing so "opens" a door for the user to use his or her MVS identity to legitimately manipulate data in the data set. These data could be actual data, FSP, or programs that the user would not be authorized to modify using his or her z/OS UNIX identity.

In a sysplex environment, the HFS files may be shared between members of the sysplex. But certain system directories such as /tmp for temporary files cannot be shared among sysplex members, and thus dedicated file systems are required in each member of the sysplex to contain such directories. Refer to *z/OS V1R8.0 UNIX System Services Planning, GA22-7800*, for further information about HFS file sharing in a sysplex.

6.1.3 Mount security

To dynamically add file systems, use the MOUNT command specifying the mount point and the data set name to be mounted. The file system has a specific UNIX security environment that can be specified also at mount time. There are three optional parameters for the MOUNT command (the default value is shown in italics):

MODE(READ/RDWR): READ limits access to read-only operations for the files and directories in the mounted file system.

SETUID/NOSETUID: SETUID specifies that the setuid and setgid attributes in the file security packets are to be honored. We explain setuid and setgid in 6.2.1, "The file security packet" on page 77.

SECURITY/NOSECURITY: NOSECURITY specifies that there is free access to the files and directories mounted in the file system, provided the user has search access to the mount point directory and the file system MODE is RDWR. The files extended attributes such as "a" (APF) and "p" (program control) are not honored. File access auditing is however still operating. All new files created in the file system will be owned by UID(0), whatever the UID of the actual file creator is.

MOUNT requires the requestor to have superuser authority or access to the SUPERUSER.FILESYS.MOUNT profile in the UNIXPRIV class. UPDATE access to the profile is required to specify SETUID, while READ access is sufficient for NOSETUID (keep in mind that SETUID leads to execute code under an identity different from the caller's identity when the setuid or setgid file attributes are set).

Also note that there is no privilege check for the identity that issues the MOUNT that would be related to the data set resource. Having superuser privilege or being permitted to SUPERUSER.FILESYS.MOUNT gives MOUNT authority for any data set the kernel address space is permitted access to. It is quite important to ensure that MOUNT authority is given to duly trusted users as these users can unmount an existing file system and mount it at a different mount point with NOSECURITY.

6.2 UNIX files and directories security

In this section, we address the security information that is specifically assigned to z/OS UNIX files and directories.

6.2.1 The file security packet

Each z/OS UNIX file and directory has a set of information called the *file security packet* (FSP) associated with it, which keeps access control permissions and other relevant file specific information. The FSP resides with its file or directory in the file system data set. It is created when a file or directory is created, and is deleted when the file or directory is deleted.

Note that part of the FSP can also be exported with the file when utilities such as pax and tar are used. With the consequence that it can also be imported with a file when using these utilities. It is then important to make sure that this imported FSP:

- ▶ Includes permission bits that match the security policy of the environment the file is imported into
- ▶ Contains UIDs and GIDs that are defined in the receiving system

Figure 6-3 on page 78 shows the information stored in the FSP.

Important: RACF is used to control and audit accesses to z/OS UNIX files or directories; however, there are no profiles in the RACF database that define the UNIX resources. RACF is provided with the FSP and the user security packet (USP), which contains the requestor's identity, when it comes to make an access control decision, as shown in Figure 6-2 on page 78.

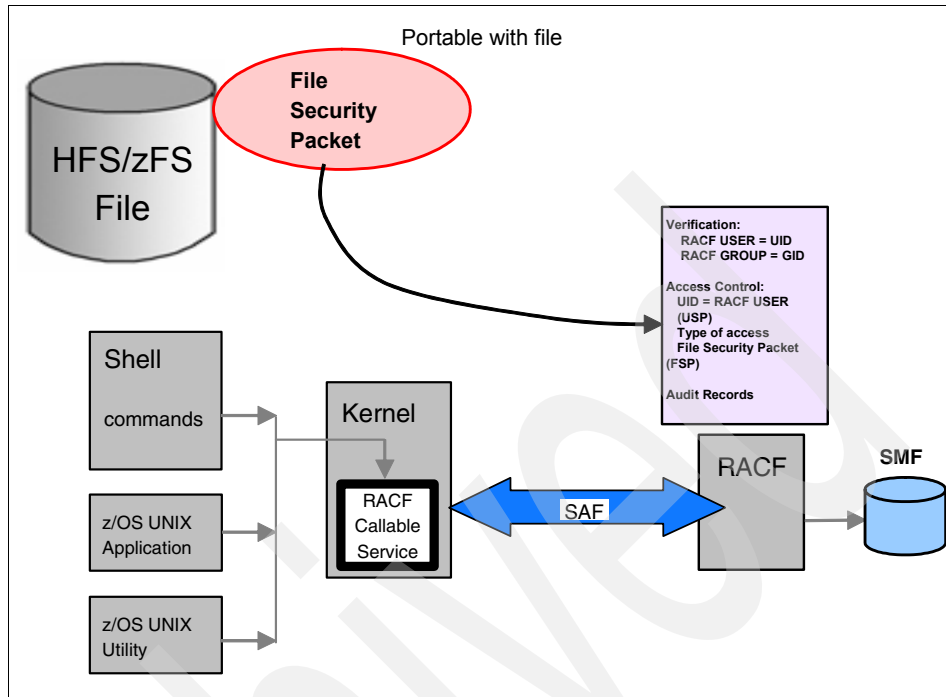


Figure 6-2 RACF and the file security packet

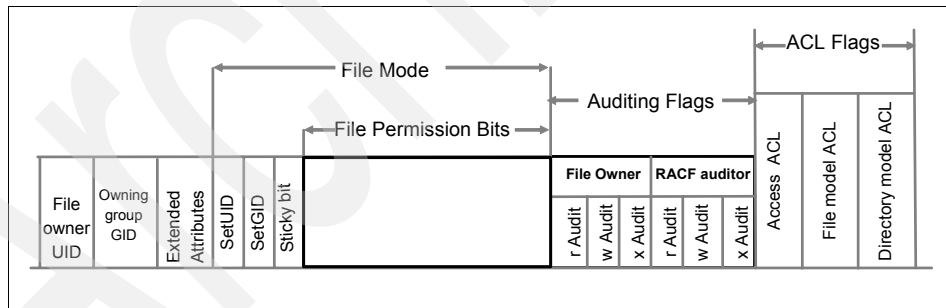


Figure 6-3 The z/OS UNIX file security packet

The following information is found in the FSP.

Owning user and owning group

The owning z/OS user UID and GID information is always present. When the corresponding setuid bits, setgid bits, or both are set in the file mode bits and the file is an executable program, then it will be executed under the UID, GID, or both

designated as the owning UID and GID, as opposed to the requestor's UID and GID. As mentioned earlier, this requires the file system to be mounted with SETUID.

These values can be set or changed using the **chown** or **chgrp** commands in the z/OS shell or from the ISPF ISHELL menu.

Extended file attributes

There are four extended file attributes:

- ▶ The “p” bit for program controlled
- ▶ The “a” bit for APF-authorized
- ▶ The “s” bit for the ability to run in multi-process (shared) address space
- ▶ The “l” bit to indicate that the program is loaded from the shared library region

These values can be set or changed using the **extattr** command in the z/OS shell or from the ISPF ISHELL menu.

The setuid and setgid bits

Having the setuid or setgid bit on indicates that when spawned, this executable runs under the UID, the GID, or both, specified as owning the file.

These values can be set or changed using the **chmod** command in the z/OS shell or from the ISPF ISHELL menu.

The sticky bit

The sticky bit indicates to the operating system that the file name points to an executable program that must first be searched according to the standard z/OS search sequence (STEPLIB/JOBLIB/LPA/LINKLIST), before looking into the z/OS UNIX file itself. If a file has both the sticky bit and the APF attribute and is found in the standard z/OS search sequence, then the file extended attribute “a” (APF) is ignored, and the APF properties of the z/OS library takes precedence. If the library is an APF library and the module is link-edited with AC=1, then it runs APF-authorized.

For a directory, the sticky bit indicates that the user attempting to delete the contents of the directory must also own the file, or own the directory, or be superuser.

This value can be set or changed using the **chmod** command in the z/OS shell or from the ISPF ISHELL menu.

File or directory permission bits

The permission bits specify who have read, write, and execute/search authority to the associated file or directory. The permission bits are further discussed in 6.3, “File and directory access control permission bits” on page 81.

The permission bits are specified in three subgroups (called *classes*) of three bits each:

- ▶ The read, write, and execute bits (rwx) for the owning UID
- ▶ The rwx bits for the owning GID
- ▶ The rwx bits for other users who are neither owners or in the owning group of the file.

Note that these permission bits are sometimes referred to as the *base ACL entries*.

These values can be set or changed using the **chmod** command in the z/OS shell or from the ISPF ISHELL menu.

Optional access control list

The user can also define up to 1024 extended access control list (ACL) entries in the file security packet, which specifies each the rwx authority for a specific UID or GID. The ACL entries are inspected in addition to the file mode permission bits when the FSSEC RACF class is active. ACLs are further discussed in 6.4, “File and directory access control: Access control list” on page 86.

These values can be set or changed using the **setfac1** command in the z/OS shell or from the ISPF ISHELL menu.

Audit attributes

There are two sets of audit attributes in the FSP: One controlled by the z/OS UNIX owner of the file or directory or a superuser, and one controlled by users with the RACF AUDITOR attribute.

To modify attributes in the AUDITOR section, the user needs to have the AUDITOR attribute in RACF and then can use the -a option on **chaudit**. The attributes tell whether successful and/or unsuccessful access attempts are audited for read, write, and execute/search access respectively. *Audited* means that RACF creates an SMF type 80 record whenever the auditing criteria are met.

Note: Auditing of successful accesses using the permission bits is bypassed when the FACILITY BPX.SAFFASTPATH profile is defined. In that case, the z/OS UNIX kernel makes the access decision itself without calling RACF. In case of unauthorized access, the kernel calls RACF to generate auditing data if required.

If you have a very sensitive file where you want all accesses to be audited, even for superusers accesses, then you may not have BPX.SAFFASTPATH defined.

Tip: The `find` shell command has many parameters that pertain to the searching files and directories for specific security-related attributes.

6.3 File and directory access control permission bits

The *permission bits* in the FSP are shown in Figure 6-4.

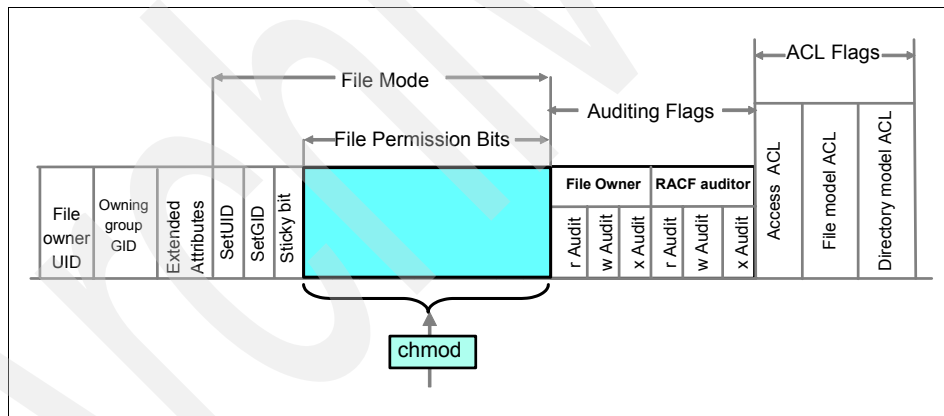


Figure 6-4 Permissions bits in FSP

The three classes of permission bits are:

- ▶ The file/directory owner r/w/x permissions: They specify the access privileges for the UID that owns the file.
- ▶ The file/directory owning group r/w/x permissions: They specify the access privileges for members of the owning GID. When “list-of-groups” checking is active in RACF (it is turned on with the command SETROPS GRPLIST), then the system examines all RACF groups the user is connected for a matching GID.

- ▶ Others r/w/x permissions, which is actually the default access for those users who do not have either the owning UID or the owning GID.

To change the permission bits for a file or a directory the user can use either the ISPF shell or the **chmod** (or **setfac1**) commands. The user issuing the command must either:

- ▶ Be a superuser
- ▶ Have the owning UID of the file, or of the directory
- ▶ Have READ access to the SUPERUSER.FILESYS.CHANGEPERMS profile in the UNIXPRIV class

The meaning of the “r/w/x” bits differs depending on whether they pertain to a file or a directory, as shown in Figure 6-5.

Access	Files	Directories
Read (r)	Read or print the contents of a file.	Read, but not search a directory.
Write (w)	Change a file, adding or deleting data.	Change a directory, adding or deleting members.
Execute or Search (x)	Permission to run an executable file.	Permission to search a directory.

Figure 6-5 Permissions bits signification

Note that the “x” permission bit, which is the authorization to search a directory, allows to use the directory in the specification of a path to access a file or another directory. However, “x” alone does not allow to list a directory or to add files or subdirectories in the directory. In order to access a file, the user must have the “x” permission bit in all the directories in the file path.

Note:

- ▶ The permission bits do not establish hierarchical permissions, that is, for instance, the write permission bit does not implicitly grant read permission.
- ▶ In order to execute a shell script, both permission bits “r” and “x” are required. In fact, interpreted language has to be read before being executed.

Permission bits are often represented using octal values, which are also often used in commands syntax. Figure 6-6 shows examples of the use of the octal values.

0	--	No access
1	--x	Execute-only
2	-w-	Write-only
3	-wx	Write and execute
4	r--	Read-only
5	r-x	Read and execute
6	rw-	Read and write
7	rx	Read, write and execute

Permission bit examples:

700	owner (7=rwx)	group (0=--)	other (0=--)
755	owner (7=rwx)	group (5=r-x)	other (5=r-x)

Figure 6-6 Octal values for permission bits

Commonly used octal configurations of the permission bits are:

- ▶ 700 to give the file's or directory's owner all accesses to the file and to deny all accesses to anybody else.
- ▶ 755 gives all accesses to the file's or directory's owner and only read access to anybody else.

6.3.1 Default permission bits

Permission bits must be assigned by default when creating a new file or directory. The default values depend both on:

- ▶ The command used to create the file or directory
- ▶ The value of the user's "umask"

This is represented in Figure 6-7.

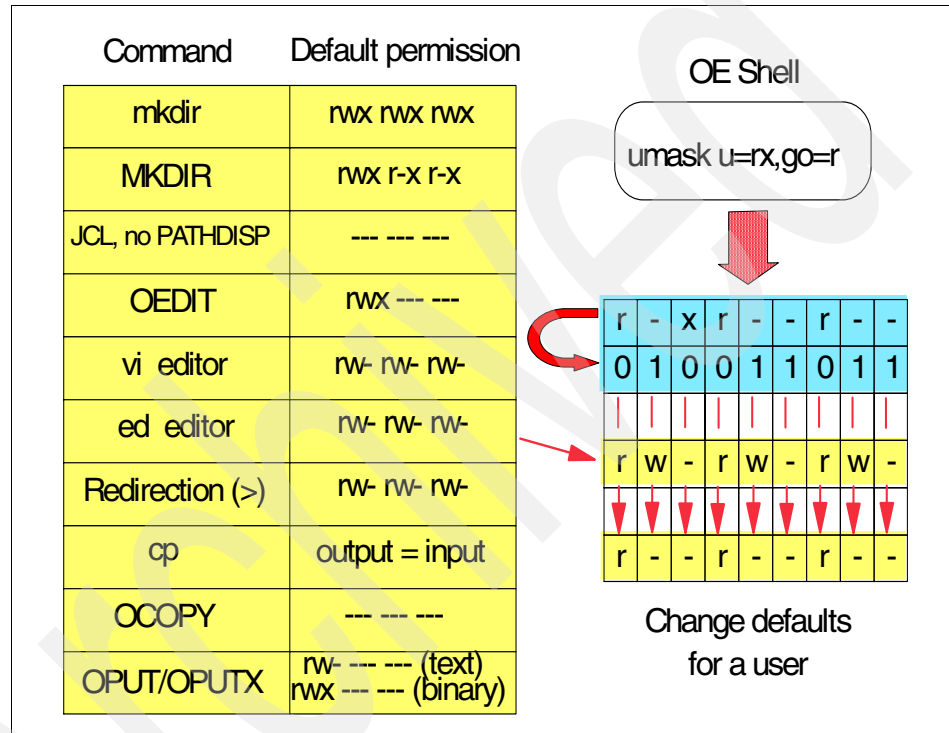


Figure 6-7 Default permission bits for files and directories

The user can change the default setting of the permission bits when a file, or directory, is created by using the **umask** shell command. The values set by the **umask** command will last for the length of the user's session, or the command can be part of the user's login so that the user always has the same default permissions.

The umask operates as follows:

- ▶ The user issues the **umask** command, which specifies allowable permissions in the classes of access.
- ▶ The actual umask, used internally in the system, is derived by converting all allowed permissions into 0 and disallowed permissions to 1, to generate an octal string.
- ▶ As an example, when the user employs the “vi” editor to create a new file, the normal set of permissions would be octal “666”. When umask is active, the permissions from “vi” are compared directly to the umask. Any permission from “vi”, which has a 1 specified in corresponding position in umask, will be *switched off*.
- ▶ Therefore, after umask processing, the user has effectively allowable permissions on the just created file.

Note: The umask only turns bits off; it cannot turn them on if the application did not specify them as on.

6.3.2 The chmod command

The **chmod** command can modify bits in either a relative way as mentioned above, or it can be absolute and reset the specified triplets and then set those selected. The operators + and - designate relative operations that leave unmentioned bits unchanged:

```
chmod -w filename
```

It clears the three write permission bits for owning UID, owning GID, and others for the file “filename”, whether they were set or not.

```
chmod a=rwx filename
```

It sets all nine permission bits and clears the setuid, segtid, and the sticky bits.

```
chmod g=rx filename
```

The above command sets the GID permissions to read and execute, and clears an eventual GID write bit, and clears the set_gid bit.

chmod -R will work on all files and subdirectories in a directory and all files and subdirectories in subdirectories to any depth.

6.3.3 Default owning UID and GID

By default, the system sets the UID and GID of the file when the file is created:

- ▶ The UID is set to the effective UID of the creating process.
- ▶ The GID is set to the GID of the owning directory. You can define FILE.GROUPOWNER.SETGID to change this behavior.

When you have defined the FILE.GROUPOWNER.SETGID profile in the UNIXPRIV class, the setgid bit for a directory determines how the group owner is initialized for new objects created within the directory.

- ▶ If the setgid bit is on, then the owning GID is set to that of the directory. It also means that the setgid bit is propagated from the parent directory to any created subdirectories.
- ▶ If the setgid bit is off, then the owning GID is set to the effective GID of the process.

Tip: When a new file system is mounted, you must turn on the setgid bit of its root directory if you want new objects within the file system to have their group owner set to that of the parent directory.

6.4 File and directory access control: Access control list

The FSP can contain optional *ACL Flags* as shown in Figure 6-8.

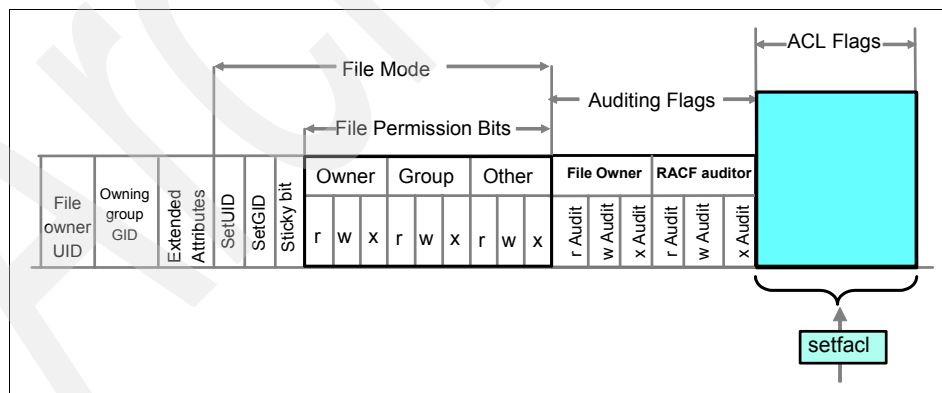


Figure 6-8 File security packet and ACL support

z/OS UNIX supports three types of ACL:

- ▶ **Access ACL:** This is the ACL used to control access to a specific file or directory. It is this ACL that is used in conjunction with the permission bits as described in 6.5.2, “Authorization checking algorithm with ACL defined” on page 92.
- ▶ **File default ACL:** This is a model ACL that files can inherit as default access ACL when created under a specific directory. It is also assigned to subdirectories as their own “file default ACL” to be propagated as the default ACL to hosted files.
- ▶ **Directory default ACL:** This is a model ACL that subdirectories inherit as default access ACL when created under a specific parent directory. The directory default ACL is copied to the created subdirectory as both its “access ACL” and “directory default ACL”.

Default ACLs are intended to help reduce the hierarchical file system administrative workload.

Important: ACLs are not inherited across mount points.

An ACL is structured as shown in Figure 6-9.

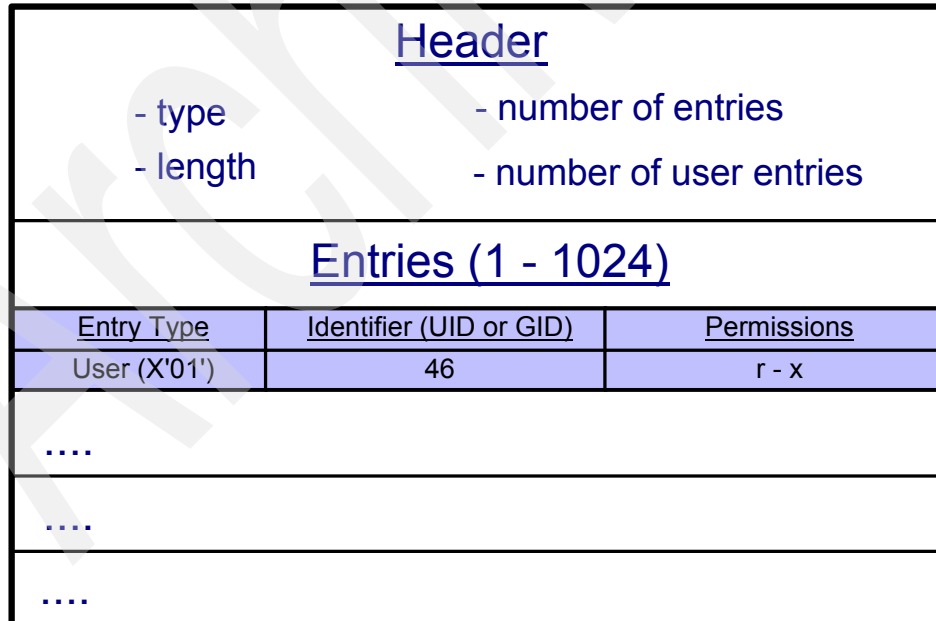


Figure 6-9 z/OS UNIX ACL structure

It consists of a list of entries, with a maximum of 1024 entries per file or directory, where every entry holds information about the type of identifier (user or group), the identifier itself (UID or GID), and permissions (read, write, and execute) associated to the identifier. The entries are sorted in ascending order by UID, then GID, to optimize the access checking algorithm.

Notes:

- ▶ ACL entries are used for access control only if the RACF FSSEC class is active. The class can be activated by the following RACF command:
`SETROPTS CLASSACT(FSSEC)`
- ▶ ACL entries are automatically deleted when the file/directory is removed.
- ▶ pax and tar support archiving ACLs along with files.
- ▶ Because z/OS ACLs can grant and restrict access, the use of ACLs is not UNIX 95-compliant (according to the X/Open UNIX 95 specification, additional access control mechanisms may only restrict the access permissions that are defined by the file permission bits).

To change entries in an ACL, you can use either the ISPF shell or the **setfac1** command. In order to do so, the requestor must either:

- ▶ Be a superuser
- ▶ Have the UID of the file, or directory, owner
- ▶ Have READ access to the SUPERUSER.FILESYS.CHANGEPERMS profile in the UNIXPRIV class

To display the ACL entries, you can use either the ISPF shell or the **getfac1** command.

6.5 File and directory access control: Security checks

When an access control decision is needed, the z/OS UNIX kernel calls the external security manager through the System Authorization Facility (SAF) interface, and supplies both the USP and the FSP, shown in Figure 6-10. The external security manager can then assess the user's authorization to access the file or directory.

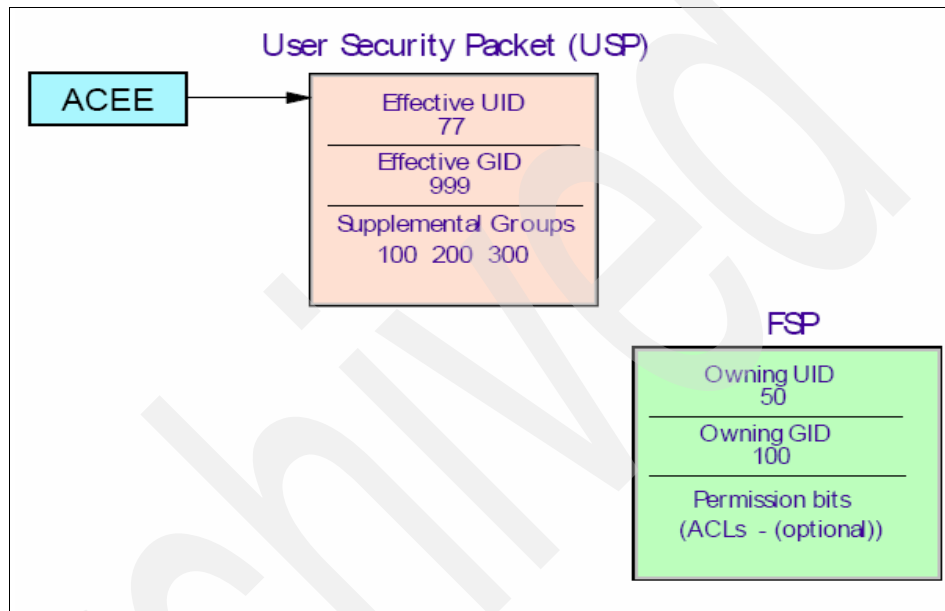


Figure 6-10 Security control blocks

The Accessor Environment Element (ACEE) is a control block that contains a description of the current user's security environment, including the user ID, current connect group, user attributes, and group authorities. The ACEE has been constructed during the user identification and authentication by RACF.

This information is used to create another control block named USP. The effective UID and effective GID of the process are used in determining access decisions (refer to 4.2, "Identities associated with a z/OS UNIX process or thread" on page 48 for the explanation of the effective UID or GID).

6.5.1 Authorization checking algorithm without using an ACL

The decision algorithm flow is shown in Figure 6-11.

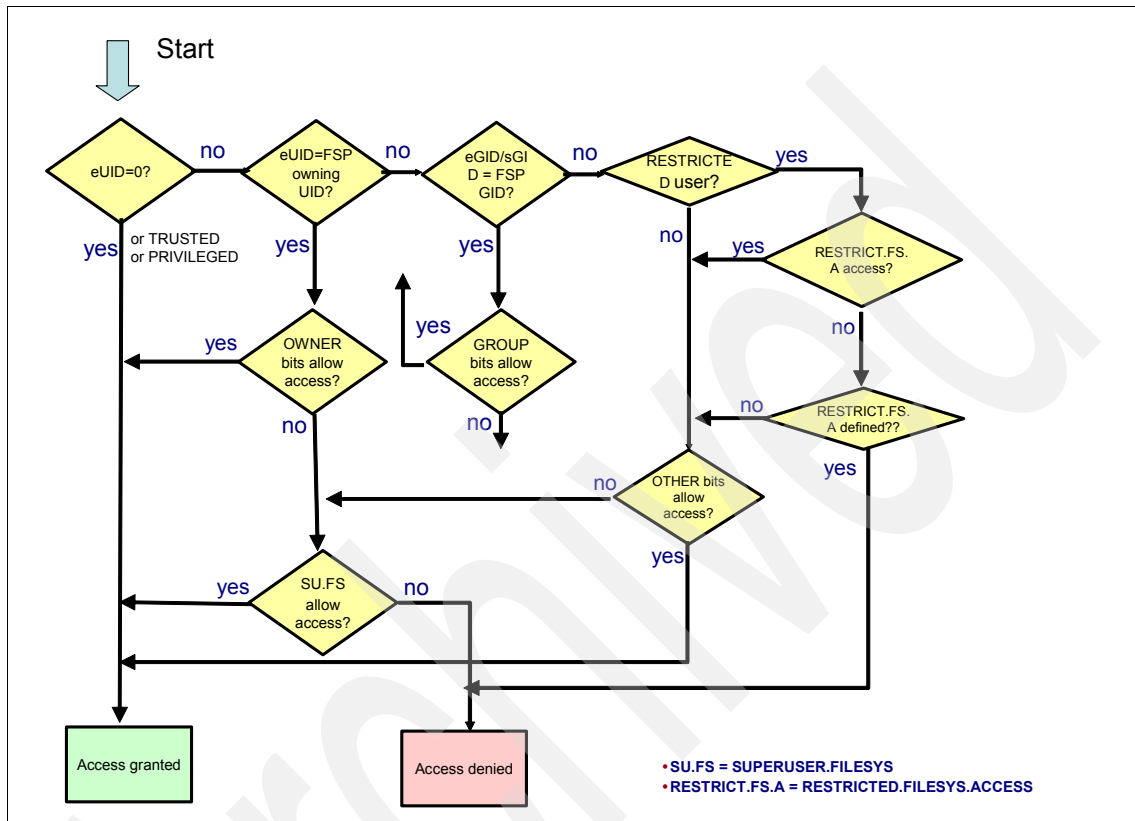


Figure 6-11 Authorization checking flow without ACL

- ▶ A superuser is allowed access to all resources, with one exception for the execution of a file: If the user, with any UID including UID(0), is requesting execute access to the file, access is always denied if none of the permissions bits grant execute access, and, if an ACL is present and the FSSEC class is active, no ACL entry grants execute access as well to the file.

Note: Tasks running with the TRUSTED or PRIVILEGED attribute do have superuser privileges even if they are running with a non-zero UID.

- ▶ If the effective UID of the process (the accessor) in the USP equals the file's owning UID, RACF uses the *owner* permission bits in the FSP to either allow or deny access.

- ▶ If the effective GID of the process in the USP equals the owning GID of the file, RACF uses the *group* permission bits in the FSP to either allow or deny access. If RACF list-of-groups checking is active, RACF will look at the user's connect groups that have a GID (“supplemental groups”) for a group that matches the owning GID of the file. If it finds a matching GID, RACF will allow or deny access based on the group permission bits specified in the FSP. (Note that if a user is connected to more than 300 z/OS UNIX groups, only the first 300 will be used.)
- ▶ If the effective UID or GID of the process does not match the file owning UID or GID, then the *others* permission bits determine access. See the possible effect of the RESTRICTED.FILESYS.ACCESS profile in this case as described below.
- ▶ If access is denied by the permission bits but if the user ID is permitted to the profile SUPERUSER.FILESYS in the UNIXPRIV class, the user is granted access to any file or directory.

RESTRICTED.FILESYS.ACCESS

This checking is done for users defined with the RESTRICTED attribute in their RACF user profile (RESTRICTED users do not have access by default to MVS resources, that is, they need to be explicitly granted access in the resource access list). The RESTRICTED.FILESYS.ACCESS profile, when defined, enforces the RESTRICTED user concept for z/OS UNIX access as well: RESTRICTED users are not getting the default authorization provided by the FSP others permission bits. This restriction is lifted for RESTRICTED users, or one of the groups they belong to, that are permitted in READ to the RESTRICTED.FILESYS.ACCESS profile. In this latter case, the user's access is still controlled by the others permission bits.

SUPERUSER.FILESYS

The SUPERUSER.FILESYS profile in the UNIXPRIV class has three access levels that allow access to z/OS UNIX files as follows:

- ▶ **READ:** Allows a user to read any local file, and to read or search any local directory.
- ▶ **UPDATE:** Allows a user to write to any local file, and includes privileges of READ access.
- ▶ **CONTROL/ALTER:** Allows a user to write to any local directory, and includes privileges of UPDATE access.

Note: Figure 6-11 on page 90 is missing two additional specific paths in this decision algorithm:

- ▶ A RACF AUDITOR can read and search any directory.
- ▶ For threads running with unauthenticated client, both client and server must be authorized to the file if the BPX.SERVER profile is defined, as explained in 5.5, “BPX.SERVER” on page 62. The same algorithm is applied to both client and server.

6.5.2 Authorization checking algorithm with ACL defined

If an ACL is defined and the FSSEC RACF class is active, then the decision tree shown on Figure 6-12 is executed.

Below Figure 6-12, we provide a description of the different branches and nodes of this algorithm.

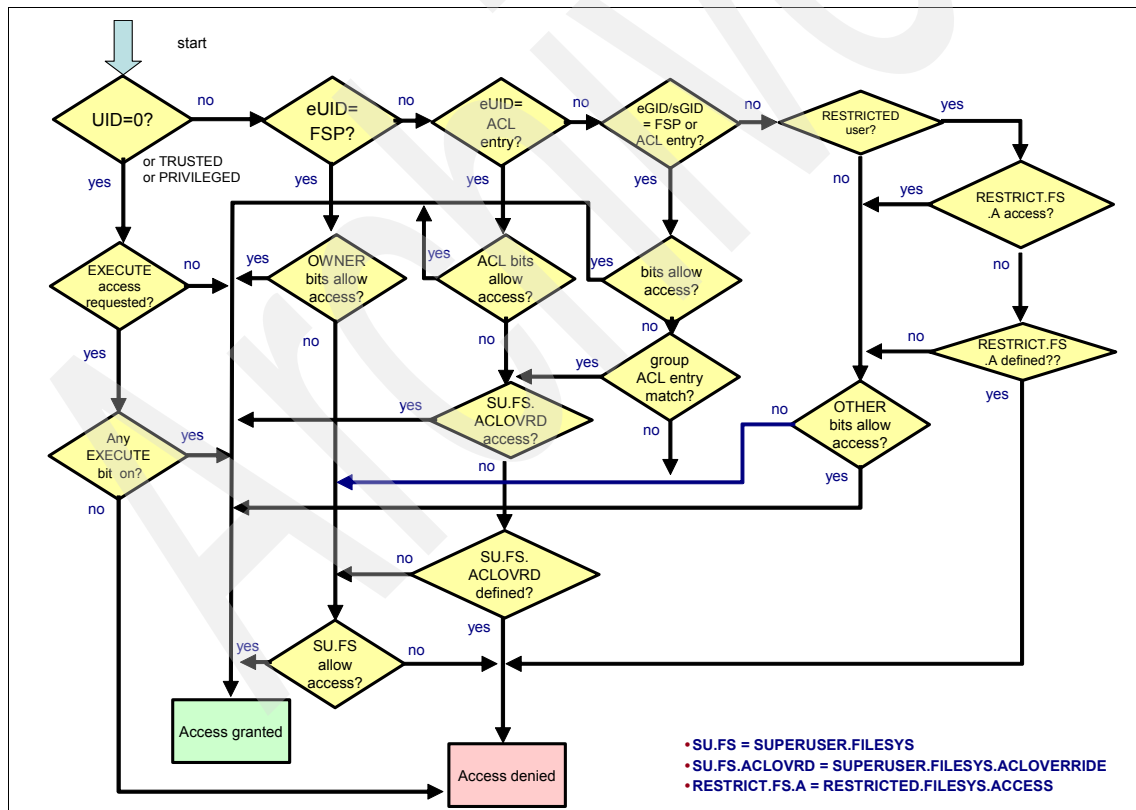


Figure 6-12 Authorization checking flow with ACL

1. UID checking

If the user is not a superuser, the permission bits are first checked:

- If the requestor's UID matches the file owning UID, the owner permission bits are checked. If the access requested is not allowed, the requestor's permission to the SUPERUSER.FILESYS profile is then checked.
- If the requestor's effective UID does not match the file owning UID, the ACL entries are checked. If the selected requestor's UID matches an ACL entry, the ACL entry bits are checked. If the requested access is not allowed in the ACL entry, a check is done to the SUPERUSER.FILESYS.ACLOVERRIDE profile in the UNIXPRIV class (see below for an explanation of this profile), to determine if the access denied by the ACL primes over the user's permission to SUPERUSER.FILESYS.

2. GID checking

If a matching ACL entry is not found for the requestor's UID user, the requestor's current connect group, and supplemental groups (if list-of-groups is in effect), are checked for:

- If the requestor's group GID matches the file owning GID, the group permission bits are checked. If the group permission bits allow the requested access, then access is granted.
- If any of the requestor's supplemental GIDs matches the file owning GID, the group permission bits are checked. If the group permission bits allow the requested access, then access is granted.
- If no group permission bits access is allowed and there are ACL entries for the requestor's GID or any of the supplemental GIDs, then the permission bits of these ACL entries are checked (if the RACF FSSEC class is active). If one ACL entry allows the required access then access is granted. If the ACL entries that match the requestor's GID do not allow the access, then SUPERUSER.FILESYS.ACLOVERRIDE is checked as indicated below.

3. Check SUPERUSER.FILESYS.ACLOVERRIDE

This checking is done only when a user's access was denied by an ACL entry matching the user's UID or one user connect group's GID. The purpose of the profile SUPERUSER.FILESYS.ACLOVERRIDE in the UNIXPRIV class is to get the ACL entry denial of access decision enforced even if the user has an access on the profile SUPERUSER.FILESYS in the UNIXPRIV class (actually this works similarly to the OPERATIONS attribute when given to an MVS user: The user has access to all data sets unless explicitly denied access in the data set access list).

If the SUPERUSER.FILESYS.ACLOVERRIDE profile has been defined and it is established that it should not apply for some users, then these users or

groups can be permitted READ, UPDATE, or CONTROL to SUPERUSER.FILESYS.ACLOVERRIDE. (Actually the same access level as would have been required for SUPERUSER.FILESYS.) As shown in Figure 6-12 on page 92, these users' access request, if not denied with SUPERUSER.FILESYS.ACLOVERRIDE, will next go through the permission checking to SUPERUSER.FILESYS.

4. Check SUPERUSER.FILESYS

If the user ID is permitted to the profile SUPERUSER.FILESYS in the UNIXPRIV class, then the user is always permitted to access files or directories, as a superuser would be, to the extent however of the access level given to SUPERUSER.FILESYS, as explained in 5.7.1, "The UNIXPRIV class of resources" on page 65.

5. Check RESTRICTED.FILESYS.ACCESS

If no match was found in the permission bits or ACL entries, then the other class permission bits is checked, unless the requestor has the RESTRICTED attribute and the profile RESTRICTED.FILESYS.ACCESS in the UNIXPRIV class is defined. If the requestor's user ID, or one of the requestor's group, is not permitted to RESTRICTED.FILESYS.ACCESS, then the RESTRICTED status of the user is enforced for z/OS UNIX files and directories as well.

Important: The *z/OS V1R8.0 Security Server RACF Administrator's Guide*, SA22-7683, has an appendix dedicated to the debug of access control setups. This appendix thoroughly describes the access control algorithms for z/OS UNIX files and directories and provides extremely useful debug guidance.

6.6 The IRRHFSU utility

The IRRHFSU utility is available as a network download at:

<http://www.ibm.com/servers/eserver/zseries/zos/racf/irrhfsu.html>

The utility unloads the UNIX System Services Hierarchical File System file security information in a manner compatible with IRRDBU00 utility. RACF provides the IRRDBU00 utility to unload the contents of the RACF database into a flat file suitable for viewing or loading into a relational database for querying. Similarly, the IRRHFSU utility downloads data contained within the FSP (such as file permission bits, owning UID and GID, owner-specified and auditor-specified logging options) into a UNIX file or an MVS data set.

The **find** command can locate files with ACLs containing *orphaned* ACL references, that is, entries for UIDs and GIDs that cannot be mapped to RACF user or group profiles. However, the **find** output is not useful for removing these references, because the UID or GID is not reported as part of the output. The IRRHFSU utility can be invoked with a parameter, which results in deletion of orphaned ACL entries.

For more information, see:

<http://www-1.ibm.com/servers/eserver/zseries/zos/racf/goodies.html>

Archived

Archived

Overview of multilevel security

Multilevel security (MLS) is an optional security model with its base features implemented in RACF since 1990. However, at this time, MLS did not address specific resources such as the TCP/IP resources or the UNIX System Services resources. This is addressed in enhancements to RACF MLS at z/OS V1R5 and z/OS V1R6. In this chapter, we introduce, at a very high level, the principles of operation of an MLS system and how they can be used to protect z/OS UNIX resources.

Refer the following books for the setup and use of MLS:

- ▶ *z/OS V1R8.0 Security Server RACF Security Administrator's Guide*, SA22-7683
- ▶ *z/OS Planning for Multilevel Security and the Common Criteria*, GA22-7509

7.1 The MLS security model

Multilevel security is a security policy that allows the classification of data and users on a system of hierarchical security levels combined with a system of non-hierarchical security categories. This is shown in Figure 7-1 where resources and users are classified as belonging to categories and a security level. The security level implies a hierarchy in data sensitivity, while the categories are an arbitrary differentiation between data or users.

When using MLS, there are actually two layers of access control being used:

- ▶ The *mandatory access control* (MAC) that locates users and resources in terms of security level and category and then applies the MAC access control rules as explained in 7.1.1, “Applying MAC”.
- ▶ If the MAC access control allows access, then the regular access control mechanism, using RACF resource profiles or z/OS UNIX permission bits and ACLs, is used. Note that this access control mechanism is termed as *discretionary access control* (DAC) in the MLS terminology.

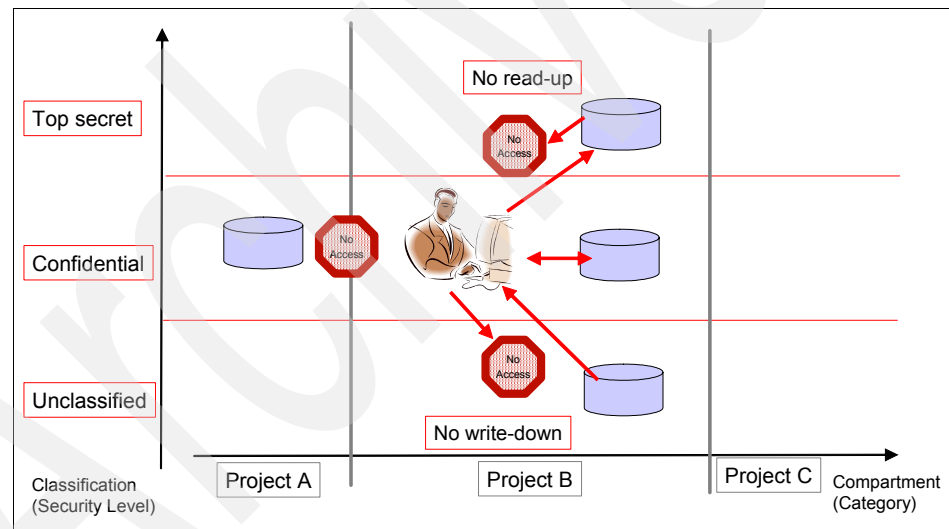


Figure 7-1 The MLS security model

7.1.1 Applying MAC

A simplified description of the access control rules with MAC is:

- ▶ Users not in the same categories as resources will never get access to these resources. Note that a specific user or resource can be defined as being in several categories.

- ▶ When a user and a resource are in the same category, the user has the following accesses, if also authorized by DAC:
 - Read-only access to resources at a lower security level: The idea is that the user cannot then declassify data by writing them at a confidentiality level lower than his or her own security level.
 - Write-only access to resources at a higher security level.
 - Read and write access to resources at the same security level.

Practically, the users and resources categories and security levels are specified through security labels.

7.1.2 Security labels

The categories to be used for MAC checking are described in the CATEGORY profile in the RACF SECDATA class. Categories are defined as members of the CATEGORY profile:

```
RDEFINE SECDATA CATEGORY UACC(NONE)
RALTER SECDATA CATEGORY ADDMEM(PROJECTA, PROJECTB, PROJECTC)
```

The security levels are defined in the SECLEVEL profile in the RACF SECDATA class. The security levels are members of the SECLEVEL profile and are designated with a name and a number, the latter is used to establish the sensitivity level of the seclevel:

```
RDEFINE SECDATA SECLEVEL UACC(NONE)
RALTER SECDATA SECLEVEL ADDMEM(UNCLASSIFIED/10, CONFIDENTIAL/20, +
TOPSECRET/30)
```

Then security and categories are compounded into security labels (do not confuse with security *levels*), which are profiles in the SECLABEL class. A SECLABEL profile is defined with a name and two components:

- ▶ The names of the categories comprised in this security label (one or more categories)
- ▶ The name of the security level that this security label refers to (only one security level):

```
RDEFINE SECLABEL EAGLE SECLEVEL(TOPSECRET) +
ADDCATEGORY(PROJECTA,PROJECTB)+ UACC(NONE)
RDEFINE SECLABEL SPARROW SECLEVEL(UNCLASSIFIED) UACC(NONE)
```

Then users are assigned security labels, actually they are given access to SECLABEL profiles:

```
PERMIT security-label CLASS(SECLABEL) ACCESS(READ) ID(user-id-1 +
user-id-2 ...)
```

And they are also specified a default security label in the USER profile:

```
ALTUSER user-id SECLABEL(default-seclabel)
```

Note that the user must also be permitted to the profile of its default SECLABEL.

Finally resources are also given a security label (only one):

```
ALTDSD 'dataset-profile' SECLABEL(security-label)
```

7.1.3 Domination and equivalence of security labels

Security labels are used for MAC according to rules of domination or equivalence between the accessor and the resource security labels.

Domination

For security label A to dominate security label B, the security level of A must be higher or equal to the security level of B, and A has at least all the categories specified in B.

A user who dominates a resource can read the resource. A user who is dominated by a resource can write into the resource.

Equivalence

For security label A to be equivalent to security label B, the security level of A must be equal to the security level of B, and both A and B have the same set of categories.

A user who has an equivalent security label to the resource can read and write the resource.

Disjoint security labels

Otherwise, the security labels are *disjoint* or *incompatible* and the user cannot get any access to the resource.

7.1.4 Turning on MLS in RACF

There are a set of SETROPTS options that turn MLS on and provide a fine control of the mechanisms. They are further explained in the reference documents:

- ▶ COMPATMODE and NOCOMPATMODE
- ▶ MLACTIVE and NOMLACTIVE
- ▶ MLQUIET and NOMLQUIET
- ▶ MLS and NOMLS
- ▶ MLSTABLE and NOMLSTABLE
- ▶ SECLABELAUDIT and NOSECLABELAUDIT
- ▶ SECLABELCONTROL and NOSECLABELCONTROL
- ▶ SECLEVELAUDIT and NOSECLEVELAUDIT
- ▶ MLFSOBJ
- ▶ MLIPCOBJ
- ▶ MLNAMES and NOMLNAMES
- ▶ SECLBYSYSTEM and NOSECLBYSYSTEM
- ▶ SECLABELAUDIT and NOSECLABELAUDIT

7.1.5 When MLS is on

When MLS is on in RACF, the following miscellaneous controls are applied to accesses to resources, some of them depend on whether the SETROPTS options listed above are active or not:

- ▶ The system controls access to resources
 - Mandatory access control (MAC)
 - Discretionary access control (DAC)
- ▶ The system does not allow a storage object to be reused until it is purged of residual data.
- ▶ The system enforces accountability by requiring each user to be identified, and creating audit records that associate security-relevant events with the users who cause them.
- ▶ The system labels all hardcopy with security information.
- ▶ The system optionally hides the names of data sets, files, and directories from users who do not have access to those data objects.
- ▶ The system does not allow a user to declassify data by “writing down” (that is, write data to a lower classification than the classification at which it was read) except with explicit authorization to do so.

7.2 MLS and z/OS UNIX resources and users

Security labels can be assigned to z/OS UNIX files and directories and, when MLS is on, are checked against the security labels allocated to the z/OS UNIX users (actually the SECLABEL profiles the users are permitted to).

The zFS file system is the only physical file system with support for security labels in a multilevel-secure environment. The Hierarchical File System does not provide support for security labels in a multilevel-secure environment. There is setup that can allow for the use of HFS file systems in this environment, but capability is limited to read-only access.

Traditionally, access to z/OS UNIX resources is based on POSIX permissions. With the SECLABEL class active, authorization checks are performed for security labels in addition to POSIX permissions, to provide additional security. Security labels are used to maintain multiple levels of security within a system. By assigning a security label to a resource, the security administrator can prevent the movement of data from one level of security to another within the z/OS UNIX environment. When the SECLABEL class is active, security labels can be set on z/OS UNIX resources in the following ways:

- ▶ When a physical file system or zFS aggregate is created, the file system root will be assigned the security label that is specified in the RACF data set profile that covers the data set name. If a security label is not specified or if a data set profile does not exist, then a security label will not be assigned to the file system root.
- ▶ zFS file systems support the **ch1abe1** utility, which allows the setting of an initial security label on a file or directory. Use this utility to set security labels on zFS files and directories after they have been created.
- ▶ If a directory has been assigned a security label through one of the above steps, then new files and directories created within that directory will inherit a security label as follows:
 - If the parent directory is assigned a security label of SYSMULTI, the new file or directory is assigned the security label of the user. If the user has no security label, no label is assigned to the new object.
 - If the parent directory is assigned a security label other than SYSMULTI, the new file or directory is assigned the same security label as the parent directory.
 - The rules for security label assignment are more extensive when running in a multilevel-secure environment.

Notes:

- ▶ SYSMULTI is a system built-in security label that is always going to be equivalent to any other security labels.
- ▶ The SETROPTS options MLFSOBJ and MLIPCOBJ are used to turn MLS on or off respectively for files and directories and for Interprocess communications (IPCs).

Archived

Archived



Considerations on z/OS UNIX program management

In this chapter, we focus on programs residing in z/OS UNIX files and give miscellaneous considerations that pertain to the security attributes of these programs. Note that most of the information provided in this chapter was already mentioned in previous chapters, but we believe it is useful for the reader to review it in this specific context.

8.1 How to link-edit program into HFS files

Here is a sample assembly and linked batch job step for placing a load module in a Hierarchical File System (HFS) file:

```
// EXEC ASMACL, PARM.C=(NODECK),  
// PARM.L='XREF,LET,LIST,XCAL,AMODE=31,RMODE=ANY'  
//C.SYSRINT DD SYSOUT=*  
//C.SYSPUNCH DD DSN=ITSOFTC.USS.OBJ(FORK),DISP=SHR  
//C.SYSLIB DD DSN=SYS1.MACLIB,DISP=SHR  
//          DD DSN=SYS1.MODGEN,DISP=SHR  
//C.SYSIN DD DISP=SHR,DSN=ITSOFTC.USS.SRCE(FORK)  
//L.SYSLMOD DD PATH='/u/itsoftc/fork',  
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU  
//L.SYSLIB DD DSN=SYS1.CSSLIB,DISP=SHR
```

The APF rules for programs that reside in the HFS are similar to those for programs that reside in MVS authorized libraries. Setting the APF-authorized extended attribute bit should be thought of as putting that program into an authorized library. If you try to run a program from an authorized library that is not linked AC=1, it will not run APF-authorized, but that same program can be fetched by another that is running APF-authorized and executed in the authorization state in which it is called, or even have its state changed.

If the specified program is going to be invoked as a job step program, you must link-edit it with AC=1. In order to avoid possible integrity problems, do not set AC=1 if the program will be run in an APF-authorized environment but not as the job step program (such as DLL).

8.2 Owner information for a z/OS UNIX file

Assuming that the FILE.GROUPOWNER.SETGID profile is defined, the owning UID and GID assigned in the file security packet (FSP) are the UID and GID of the creator of the file. For the batch job in the previous section, it is the UID and the current connect group (or the default group) GID of the user running the batch job. It can also be the default UID and GID that are provided via the BPX.DEFAULT.USER profile.

To change the owner information, the user has to be the owner, or a superuser, or a user with access to UNIXPRIV SUPERUSER.FILESYS.CHOWN, or UNIXPRIV CHOWN.UNRESTRICTED must be defined, the latter allowing all users to change ownership of their own files. Then the following command can be used to change the owner information in the FSP:

```
chown itssoftx:finn aparm
```

This commands changes, for the file *aparm*, the owning user ID to ITSOFTEX and owning group to FINN. This is also translated into the corresponding UID and GID in the FSP. The **chown** command can change either user or group or both, while **chgrp** changes only group.

Warning: Both **chown** and **chgroup** clear the setuid and setgid bits in the file mode, therefore if the executable file is supposed to run under one of the owning identities, the **chmod** command must be used to set the setuid and setgid bits again.

8.3 Extended attributes of an HFS file

The extended attributes are the following four bits, manipulated by the **extattr** command:

- ▶ **p:** The program controlled bit
- ▶ **a:** The APF bit
- ▶ **s:** The shared space bit
- ▶ **l:** The shared library region bit

8.3.1 Program control bit

The program control bit tells the operating system that this program is considered clean, therefore executing it will not make the environment dirty. It corresponds to programs matched by PROGRAM profiles with ADDMEM operands identifying libraries.

An environment becomes dirty by running a program that is not RACF program controlled. Once an environment is marked dirty, it remains dirty (z/OS has an exception for Time Sharing Option (TSO) that temporarily can regain a clean state with some constraints).

The RACF controlled programs are those the installation considers trusted, in the sense that they perform what they are supposed to perform and nothing else. In particular, these programs do not install “backdoors”, update an APF library or an IBM DB2® table or run some administrative command, just because their caller

suddenly is one with the needed authority. They can safely be used by all users on the installation, including security, storage, and database administrators and system programmers.

In practice, there is a need for trusting more than just the operating system. This could be software from other vendors than the operating system vendor, and it might even be some of the installation's own software. The program control bits is how software is marked as trusted in z/OS UNIX.

Every update of a file with the “p” bit leads to erasure of the bit. It must be reassigned using `extattr` after every update, and this impacts software maintenance procedures.

The `extattr` command requires to be owner of the file or a superuser (UID(0)) and also to be permitted to the BPX.FILEATTR.PROGCTL.

A sample command to make the z/OS UNIX executable file `aparm` to be program controlled is:

```
extattr +p aparm
```

8.3.2 APF bit

The APF bit marks a program as one that is intended to run authorized, that is, it can potentially bypass security controls and auditing.

The APF bit is also assigned using the `extattr` command by users who are the file's owner or are superusers, *and* who have access to BPX.FILEATTR.APF in the FACILITY class. If this profile does not exist, the APF bit cannot be assigned for z/OS UNIX files.

```
extattr +a aparm
```

Unless sanction lists are in use, this bit alone decides whether an executable in a z/OS UNIX file runs authorized or not. An overview of the sanction list mechanism is given in 8.5, “The sanction list” on page 111.

8.3.3 The shared space bit

Multiple z/OS UNIX processes may run in the same address space simultaneously, provided they all run under the same UID and GID, with the objective of getting better utilization of the system's resources. The “s” bit applies to executable code in a z/OS UNIX file and enforces the value given to the `_BPX_SHAREAS` environment variable (YES/REUSE or MUST).

For the benefit of performance the bit is on by default for new files.

Note however that a process sharing an address space then has the restriction that it may not use the `setuid()` service (which changes UID). If the program is a daemon or server that needs to use `setuid()`, you must remove the “s” bit with the `extattr` command so that address space sharing does not occur when the program is spawned:

```
extattr -s aparm
```

8.3.4 The library bit

The library bit indicates to z/OS UNIX that the program should not be loaded from the file, but instead the system is to get a copy of the program residing in the shared library region. Using the shared library makes the program available to many users with a very short load time. The shared library is intended mainly to hold dll like program modules.

To set the library bit, you need access to FACILITY class entity BPX.FILEATTR.SHARELIB. It is also important, from the security standpoint, to tightly control who has access to this profile as it also determines the origin of the code that is loaded into the address space and implicitly allows consumption of the system resources needed for the shared library function.

8.4 The file mode section of the FSP

As already mentioned, this topic has already been addressed in further detail, specifically in Chapter 6, “z/OS UNIX files security” on page 73. We review it with a special focus on the program management context.

8.4.1 The non-permission bits

These are the `setuid`, the `setgid`, and the sticky bit. The first two bits specify that the program must run with the owning UID or GID, and therefore the system changes the effective UID or GID during program execution. The sticky bit specifies that the operating system should look for the program using first the standard z/OS search sequence (STEPLIB, JOBLIB, LPA, LINKLIST) then, if not found, attempt to load the program from the z/OS UNIX file.

The bits are assigned, removed, or displayed with the `chmod` command. This command requires file ownership or superuser authority or to be permitted to SUPERUSER.FILESYS.CHANGEPERMS in the UNIXPRIV class. There is no RACF profile to further restrict the use of the `chmod` command.

8.4.2 The permission bits

Let us assume that a program resides in the file *pthread*s, which is owned by the user ITSOFTC. Another user ITSOFTX is in the same group as ITSOFTC, but has a different UID.

When the user ITSOFTX enters the z/OS shell with the OMVS command and invokes the program by entering `/u/itsoftc/pthreads`, then ITSOFTX is denied execute access with the following violation message in the system SYSLOG:

```
ICH408I USER(ITSOFTX ) GROUP(SYS1 ) NAME( FINN T CHRISTENSEN )
/u/itsoftc/pthreads
CL(FS0BJ ) FID(01E6D9D2F1F6D600071E000000200000)
INSUFFICIENT AUTHORITY TO OPEN
ACCESS INTENT(--X) ACCESS ALLOWED(GROUP ---)
EFFECTIVE UID(0000000099) EFFECTIVE GID(0000000000)
```

We can see that ITSOFTX has UID(99) and that the group access class could yield the requested access (we do not see anything for the “owner” and “other” access class). However, the permission bits in this class are all off and therefore access is denied.

If user ITSOFTC, while in his home directory, issues the command:

```
ls -E pthreads
```

The following output is produced that shows the permission bits currently set:

```
-rwx----- --s- 1 ITSOFTC SYS1 4096 Jun 24 18:46 pthreads
```

This confirms that read, write, and execute access are granted only for the owner of the file.

If we want to permit the owning group read, write, and execute access to the *pthread*s program, then the owner or a user with superuser authority can enter:

```
chmod g=rwx pthreads
```

```
ls -E pthreads
```

```
-rwxrwx--- --s- 1 ITSOFTC SYS1 4096 Jun 24 18:46 pthreads
```

Now the user ITSOFTX can run the original command without getting violation messages, and is able to execute the program, provided he also:

- ▶ Has at least execute access to all directories in the path to the program file
- ▶ Or is a RACF AUDITOR
- ▶ Or has superuser authority

Note that installations that want to exploit the requirement for execute access to all directories in the path to block access otherwise granted by the permission bits, may run afoul on the AUDITOR attribute. AUDITORS cannot be kept from using files whose permissions bits allow the access. Such installations should at least have a minimum number of AUDITORS.

Note: Installations that plan on blocking access to files by not providing search access for directories have to be aware that RACF AUDITORS are always permitted to search through directories. Therefore, AUDITORS cannot be prevented from using files, whose permissions bits allow the access.

8.5 The sanction list

An installation can build a list to contain the lists of z/OS UNIX path names and MVS program names that are approved by the installation for use by APF-authorized or program-controlled calling programs. This file contains properly constructed path names and program names, and has to be located in the /etc directory.

This list provides another layer of control on which programs authorized or controlled programs are actually sanctioned by the installation. Note that programs pointed at by the sanction list are not implicitly program controlled or granted APF authorization, designated z/OS executables must also be marked with the “p” or “a” extended attribute.

A sanction list is activated using the AUTHPGMLIST('/etc/<file_name>') in the BPXPRMxx member, or by dynamically adding it using the SETOMVS command.

Sanction lists can contain up to three separate lists:

- ▶ A list of directories that is only used in the execution of an hfsload (or C dlload), exec, spawn, or attach_exec from an authorized program. All path names are considered to be approved path names from which authorized programs can be invoked.
- ▶ A list of directories that is only used in the execution of an hfsload (or C dlload), exec, spawn, or attach_exec from an executable that is running program controlled. All path names are considered to be approved path names from which program controlled programs can be invoked.
- ▶ A list of program names that are allowed to get control of APF-authorized programs as a result of an exec or spawn. These names are MVS program names. All programs are considered to be approved program names that can get control APF-authorized.

If the sanction list is running on the system, you will get error messages when you try to run program-controlled or APF-authorized programs that are not in the sanction list. You will have to add them to the sanction list.

You need to know what directories and what programs are to be set into this file. You can partially construct this file and add path names and program names as you go along. A partially complete file can be activated and when additional entries are known, this file can be updated. A background task will automatically check this file every 15 minutes for updates and then incorporate them.

Only one sanction list check is done for each program invocation. Although links in directories are supported, sanction list processing only performs one check. This check uses the path name or program name that was specified by the user.

Proper controls have to be put in place to protect access to the sanction list.

Directives on how to compose and activate sanction lists are given in *z/OS V1R8.0 UNIX System Services Planning, GA22-7800*.

Auditing z/OS UNIX

In this chapter, we provide an overview of the Resource Access Control Facility (RACF) auditing options for accesses to z/OS UNIX resources. We specifically address:

- ▶ The auditing flags in the file security packet (FSP)
- ▶ The auditing classes in RACF
- ▶ The specific auditing for superusers authority
- ▶ The generation of audit reports

The reference book to get further details about these topics is *z/OS V1R8.0 Security Server RACF Auditor's Guide, SA22-7684*.

9.1 Overview of auditing options

The overall auditing infrastructure of z/OS UNIX is shown in Figure 9-1. It points out that the external security manager is expected to provide System Management Facility (SMF) records that are used to build the audit trail and reports.

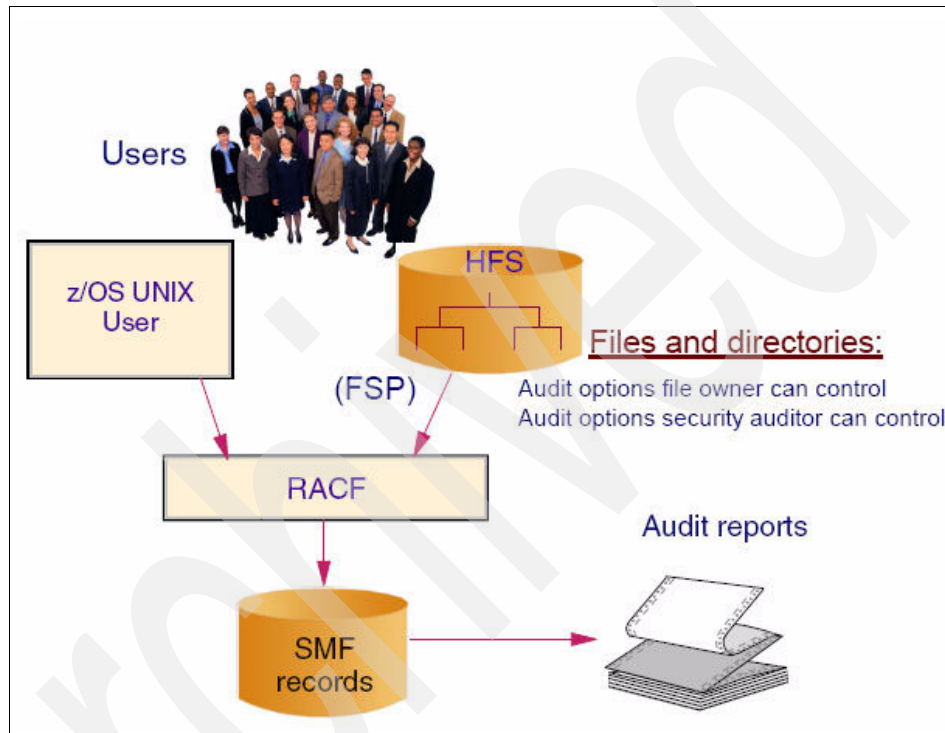


Figure 9-1 Overview of auditing options for z/OS UNIX

RACF reports audited events in SMF type 80 records, always providing full accounting of the user involved in the event by giving the user ID and UID.

For violations occurring in the UNIX System Services environment, the user's effective UID and effective GID are displayed in violation messages of the ICH408I type. These identities are used to determine the user's privilege for the intended operation. Note that they may not always match the identities defined in the relevant RACF USER and GROUP OMVS segments, because UNIX System Services provides methods by which another identity can be assumed.

9.2 File-based auditing options

The file security packet (FSP) includes the *auditing flags*, as shown in Figure 9-2, which pertain to the file or directory.

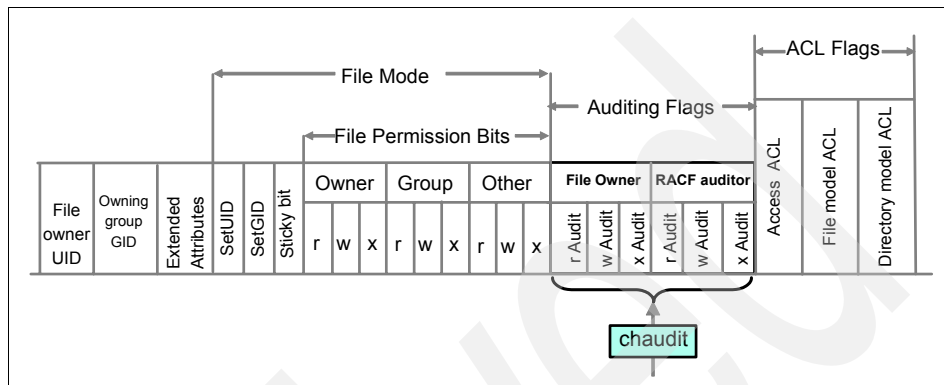


Figure 9-2 Auditing flags in the file security packet

Therefore, there are two categories of auditing flags:

- ▶ The *file owner* flags: You must be a superuser or the owner of the file (or of the directory) to specify these user audit options.
- ▶ The *RACF auditor* flags: You must have the RACF AUDITOR attribute to specify these auditor options.

It is the same command **chaudit**, with different parameters, which can change these auditing flags. Audit records are written based on the combined owner and AUDITOR settings.

Auditing is set for read, write, and execute (search for directories) for the following kinds of accesses:

- ▶ Successful accesses (s parameter in **chaudit** commands)
- ▶ Failures, that is, access violations (f parameter in **chaudit** commands)
- ▶ All, which is both successes and failures (a parameter in **chaudit** commands)
- ▶ None

If auditing is not specified for a file, the defaults are:

- ▶ For owner auditing: All failed accesses are audited
- ▶ For RACF AUDITOR auditing: There is no auditing by default

9.3 Events always audited

Audit records are always written:

- ▶ When a user who is not defined as a z/OS UNIX System Services user tries to dub a process
- ▶ When a user dubs a process using the default UID that is established via the FACILITY class profile named BPX.DEFAULT.USER (see 3.3, “Default UID and GID” on page 38)
- ▶ When an unauthorized user tries to mount or unmount a file system

There is no option to turn off this mandatory auditing.

9.3.1 RACF classes for auditing

Seven classes are defined in RACF to control auditing. The classes are predefined in the RACF class descriptor table (ICHRRCDX). No profiles can be defined in these classes; they are intended to be used only as parameters for the SETROPTS LOGOPTIONS. They do not need to be active to be used to control z/OS UNIX auditing. These classes are:

- DIRSRCH** Controls auditing of directory searches
- DIRACC** Controls auditing for access checks for read/write access to directories
- FSOBJ** Controls auditing for all access checks for file system objects except directory searches via SETROPTS LOGOPTIONS and controls auditing of creation and deletion of file system objects via SETROPTS AUDIT
- FSSEC** Controls auditing for changes to the security data (FSP) for file system objects
- PROCESS** Controls auditing of changes to the UIDs and GIDs of processes and changing of the thread limit via the SETROPTS LOGOPTIONS, and controls auditing of dubbing, undubbing, and server registration of processes via SETROPTS AUDIT
- PROCAT** Controls auditing of functions that look at data from or affect other processes
- IPCOBJ** Controlling auditing and logging of IPC security checks

Use SETROPTS LOGOPTIONS to specify logging options for all the classes associated with z/OS UNIX System Services. The auditing levels for LOGOPTIONS are:

- ▶ **ALWAYS:** All access attempts to resources protected by the class are audited. This overrides the file-level audit settings.
- ▶ **NEVER:** No access attempts to resources protected by the class are audited (all auditing is suppressed). This overrides the file-level audit settings.
- ▶ **SUCCESSSES:** All successful access attempts to resources protected by the class are audited. This is merged with the file-level audit settings.
- ▶ **FAILURES:** All failed access attempts to resources protected by the class are audited. This is merged with the file-level audit settings.
- ▶ **DEFAULT:** Auditing is controlled by the file-level audit settings in the FSP for z/OS UNIX files and directories (see 9.2, “File-based auditing options” on page 115). This auditing level is in effect when the SETROPTS LOGOPTIONS has not been issued yet.

Notes:

- ▶ Activating the classes has no effect on auditing or authorization checking, except for the FSSEC class, which is also used to enable the use of access control lists (ACLs) for z/OS UNIX access controls (see 6.4, “File and directory access control: Access control list” on page 86).
- ▶ The file-level audit settings, and their interaction with the SETROPTS settings, directly mirror the behavior of traditional RACF profile auditing.
- ▶ If users are accustomed to getting audit records and violations for unauthorized issuance of RACF commands, then they probably want to have SETROPTS LOGOPTIONS(ALWAYS(FSSEC)) being issued. Otherwise, they will not get the same level of auditing by default for unauthorized UNIX commands which alter security information, such as **chaudit**, **chown**, **chmod**, or **setfacl**.

For example, the syntax of the RACF command to activate auditing for all failed access (read/write/search) to directories is:

```
SETROPTS LOGOPTIONS (FAILURES (DIRSRCH, DIRACC))
```

In addition, you can use the SETROPTS AUDIT option to control auditing for accesses to the following classes of resources:

- ▶ FSOBJ: Creating and deleting file system objects
- ▶ IPCOBJ: Creating and deleting objects (message queues, semaphores, and shared memory segments)
- ▶ PROCESS: Dubbing or undubbing of a process

Note that you must have the RACF AUDITOR attribute to enter the AUDIT operand of SETROPTS RACF command. For example, the syntax of the RACF command to activate auditing for creating and deleting objects is:

```
SETROPTS AUDIT(FSOBJ, IPCOBJ)
```

You must have the RACF AUDITOR attribute to enter the LOGOPTIONS operand of SETROPTS RACF command.

9.4 Auditing for superuser authority and UNIXPRIV class privileges

If you use profiles in the UNIXPRIV class to control a subset of superuser authority, you can use the auditing options of these profiles for auditing the granting of the granular superuser privileges.

For example, to audit the successful access to the kill() function, granted by the SUPERUSER.PROCESS.KILL profile, set the audit options as follows:

```
RALTER UNIXPRIV SUPERUSER.PROCESS.KILL AUDIT(SUCCESS(READ))
```

Note also that the RACF SMF type 80 records have an indicator of whether authority was granted due to superuser authority.

9.5 Auditing reports

In this section, we briefly explain the reporting infrastructure shown in Figure 9-3.

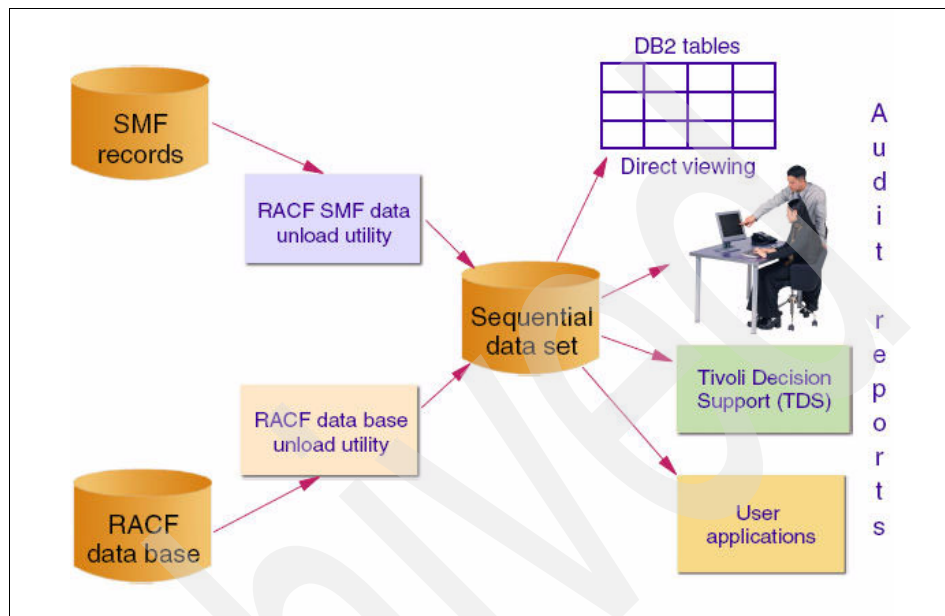


Figure 9-3 Producing audit reports

SMF records produced by RACF cannot be exploited directly. At a minimum, you must use the RACF SMF data unload IRRADU00 utility to reformat the SMF records to a sequential data set. Refer to *z/OS V1R8.0 Security Server RACF Auditor's Guide*, SA22-7684, for more information about using the IRRADU00 utility. Some sample reports are available and it is also possible to create customized reports using the RACFICE PROC (documented in same place than IRRADU00 utility).

For more elaborate reports, you will need information that is in RACF database. There is another RACF utility, named IRRDBU00, which can create in a sequential data set, several records type with the whole content of the RACF database. (Refer to *z/OS V1R8.0 Security Server RACF Administrator's Guide*, SA22-7683, for more information about using the IRRDBU00 utility.)

There are several ways to exploit the records produced by IRRADU00 and IRRDBU00 utilities:

- ▶ **User applications:** The layouts of the records are documented in *z/OS Security Server RACF library* (especially in *z/OS V1R8.0 Security Server RACF Macros and Interfaces, SA22-7682*). Based on this documentation, you can develop tools or applications to exploit these data as you want.
- ▶ **DB2 Tables:** With the IRRADU00 and IRRDBU00 utilities, RACF documentation explain how to load the produced records in DB2 tables. The DDL of several tables and some procedures are provided. When your data (events and RACF profiles) is stored in DB2 tables, you can make requests by SQL statements or develop any program to access them.
- ▶ **IBM Tivoli® Decision Support (formerly Performance Reporter):** This IBM product program has a RACF component that includes a set of DB2 tables and a lot of predefined reports to present violations or successful access on RACF resources. The reports can be graphic reports (specially for evolution or statistical reports) or tabular report (for lists of violations, for example).

Remember also that the IRRHFSU utility can be used to download data contained within the FSP (such as file permission bits, owning UID and GID, owner-specified and auditor-specified logging options) into a UNIX file or an MVS data set where they can be inspected using miscellaneous commands and tools.

BPX. RACF profiles

A number of profiles in the RACF FACILITY class have been defined for the use of z/OS UNIX, mainly used to further control the z/OS UNIX functions and attributes used by z/OS UNIX programs. Here is the list of these profiles as of z/OS V1R7.

- ▶ **BPX.CONSOLE** allows a permitted user the ability to use the `_console()` or `console2()` services.
- ▶ **BPX.DAEMON** serves two functions in the z/OS UNIX environment:
 - Any superuser permitted to this profile has the daemon authority to change MVS identities via z/OS UNIX services without knowing the target user ID's password. This identity change can only occur if the target user ID has an OMVS segment defined. If BPX.DAEMON is not defined, then all superusers (UID=0) have daemon authority. If you want to limit which superusers have daemon authority, define this profile and permit only selected superusers to it.
 - Any program loaded into an address space that requires daemon level authority must be defined to program control. If the BPX.DAEMON FACILITY class profile is defined, then z/OS UNIX will verify that the address space has not loaded any executables that are uncontrolled before it allows any of the following services that are controlled by z/OS UNIX to succeed:
 - `seteuid`
 - `setuid`

- `setreuid`
- `pthread_security_np()`
- `auth_check_resource_np()`
- `_login()`
- `_spawn()` with user ID change
- `_password()`

Daemon authority is required only when a program does a `setuid()`, `seteuid()`, `setreuid()`, or `spawn()` user ID to change the current UID without first having issued a `__passwd()` call to the target user ID. In order to change the MVS identity without knowing the target user ID's password, the caller of these services must be a superuser. Additionally, if a `BPX.DAEMON FACILITY` class profile is defined and the `FACILITY` class is active, the caller must be permitted to use this profile. If a program comes from a controlled library and knows the target UID's password, it can change the UID without having daemon authority.

- ▶ **BPX.DAEMON.HFCTL** controls which users with daemon authority are allowed to load uncontrolled programs from MVS libraries into their address space.
- ▶ **BPX.DEBUG**: Users with `READ` access to `BPX.DEBUG` can use `ptrace` (via `dbx`) to debug programs that run with `APF` authority or with `BPX.SERVER` authority.
- ▶ **BPX.FILEATTR.APF** controls which users are allowed to set the `APF`-authorized attribute in a z/OS UNIX file. This authority allows the user to create a program that will run `APF`-authorized. This is similar to the authority of allowing a programmer to update `SYS1.LINKLIB` or `SYS1.LPALIB`.
- ▶ **BPX.NEXT.USER** enables automatic assignment of UIDs and GIDs. The `APPLDATA` of this profile specifies a starting value, or range of values, from which RACF will derive unused UID and GID values.
- ▶ **BPX.FILEATTR.PROGCTL** controls which users are allowed to set the program control attribute. Programs marked with this attribute can execute in server address spaces that run with a high level of authority.
- ▶ **BPX.FILEATTR.SHARELIB** indicates that extra privilege is required when setting the shared library extended attribute via the `chattr()` callable service. This prevents the shared library region from being misused.
- ▶ **BPX.JOBNAME** controls which users are allowed to set their own job names by using the `_BPX_JOBNAME` environment variable or the inheritance structure on `spawn`. Users with `READ` or higher permissions to this profile can define their own job names.
- ▶ **BPX.SAFFASTPATH** enables faster security checks for file system and interprocess communication (IPC) constructs.

- ▶ **BPX.SERVER** restricts the use of the `pthread_security_np()` service. A user with at least READ or WRITE access to the BPX.SERVER FACILITY class profile can use this service. It creates or deletes the security environment for the caller's thread. This profile is also used to restrict the use of the BPX1ACK service, which determines access authority to z/OS resources. Servers with authority to BPX.SERVER must run in a clean program-controlled environment.

z/OS UNIX will verify that the address space has not loaded any executables that are uncontrolled before it allows any of the following services that are controlled by z/OS UNIX to succeed:

- `seteuid`
 - `setuid`
 - `setreuid`
 - `pthread_security_np()`
 - `auth_check_resource_np()`
 - `_login()`
 - `_spawn()` with user ID `change_password()`
- ▶ **BPX.SMF** checks if the caller attempting to cut a System Management Facility (SMF) record is allowed to write an SMF record. It also tests if an SMF type or subtype is being recorded.
 - ▶ **BPX.SRV.userid** allows users to change their UID if they have access to BPX.SRV.userid, where *userid* is the MVS user ID associated with the target. BPX.SRV.userid is a RACF SURROGAT class profile.
 - ▶ **BPX.STOR.SWAP** controls which users can make address spaces nonswappable. Users permitted with at least READ access to BPX.STOR.SWAP can invoke the `__mlockall()` function to make their address space either nonswappable or swappable.

When an application makes an address space nonswappable, it might cause additional real storage in the system to be converted to preferred storage. Because preferred storage cannot be configured offline, using this service can reduce the installation's ability to reconfigure storage in the future. Any application using this service should warn the customer about this side effect in their installation documentation.
 - ▶ **BPX.SUPERUSER** allows users to switch to superuser authority.
 - ▶ **BPX.UNLIMITED.OUTPUT** allows users to use the BPX_UNLIMITED_OUTPUT environment variable to override the default spooled output limits for processes.

- ▶ **BPX.WLMSEVER** controls access to the Workload Manager (WLM) server functions `_server_init()` and `_server_pwu()`. It also controls access to these C language WLM interfaces:

- `QuerySchEnv()`
- `CheckSchEnv()`
- `DisconnectServer()`
- `DeleteWorkUnit()`
- `JoinWorkUnit()`
- `LeaveWorkUnit()`
- `ConnectWorkMgr()`
- `CreateWorkUnit()`
- `ContinueWorkUnit()`

A server application with read permission to this FACILITY class profile can use the server functions, as well as the WLM C language functions, to create and manage work requests.

C/C++ functions and UNIX System Services callable services

This appendix provides the list of the C/C++ functions that can switch identity and can be protected with the BPX.DAEMON or BPX.SERVER profiles in the FACILITY class, along with the equivalent assembler services names.

- ▶ seteuid (BPX1SEU service): Set the effective UID in 31-bit mode.
- ▶ seteuid (BPX4SEU service): Set the effective UID in 64-bit mode.
- ▶ setegid (BPX1SEG service): Set the effective GID in 31-bit mode.
- ▶ setegid (BPX4SEG service): Set the effective GID in 64-bit mode.
- ▶ setuid (BPX1SUI service): Set the real, effective and saved UID in 31-bit mode.
- ▶ setuid (BPX4SUI service): Set the real, effective and saved UID in 64-bit mode.
- ▶ setuid (BPX1SGI service): Set the real, effective and saved GID in 31-bit mode.
- ▶ setuid (BPX4SGI service): Set the real, effective and saved GID in 64-bit mode.

- ▶ `setreuid` (BPX1SRU service): Set the real and/or effective UID in 31-bit mode.
- ▶ `setreuid` (BPX1SRG service): Set the real and/or effective UID in 64-bit mode.
- ▶ `setreuid` (BPX4SRU service): Set the real and/or effective UID in 31-bit mode.
- ▶ `setreuid` (BPX4SRG service): Set the real and/or effective GID in 64-bit mode.
- ▶ `_spawn` (BPX1SPN service): Invoke `spawn` with a change in user ID requested in 31-bit mode.
- ▶ `_spawn` (BPX4SPN service): Invoke `spawn` with a change in user ID requested in 64-bit mode.
- ▶ `pthread_security_np` (BPX1TLS service): Invoke `pthread_security_np()` in 31-bit mode.
- ▶ `pthread_security_np` (BPX4TLS service): Invoke `pthread_security_np()` in 64-bit mode.
- ▶ `auth_check_resource_np` (BPX1ACK service): Invoke `auth_check_resource_np()` in 31-bit mode.
- ▶ `auth_check_resource_np` (BPX4ACK service): Invoke `auth_check_resource_np()` in 64-bit mode.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this IBM Redpaper.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 128. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *ABCs of z/OS System Programming Volume 9*, SG24-6989
- ▶ *UNIX System Services z/OS Version 1 Release 7 Implementation*, SG24-7035

Other publications

These publications are also relevant as further information sources:

- ▶ *z/OS V1R8.0 UNIX System Services Planning*, GA22-7800
- ▶ *z/OS V1R8.0 MVS Initialization and Tuning Reference*, SA22-7592
- ▶ *z/OS V1R8.0 Security Server RACF System Programmer's Guide*, SA22-7681
- ▶ *z/OS V1R8.0 Security Server RACF Macros and Interfaces*, SA22-7682
- ▶ *z/OS V1R8.0 Security Server RACF Administrator's Guide*, SA22-7683
- ▶ *z/OS V1R8.0 UNIX System Services User's Guide*, SA22-7801
- ▶ *z/OS V1R8.0 UNIX System Services Command Reference*, SA22-7802
- ▶ *z/OS V1R8.0 Security Server RACF Auditor's Guide*, SA22-7684
- ▶ *z/OS V1R8.0 Network File System Guide and Reference*, SC26-7417

Online resources

These Web sites and URLs are also relevant as further information sources:

- ▶ RACF-L discussion list at: listserv@listserv.uga.edu
- ▶ z/OS UNIX discussion list at: listserv@vm.marist.edu
- ▶ IBM: z/OS information wizardry
<http://www.ibm.com/servers/eserver/zseries/zos/wizards/>
- ▶ IBM RACF: The HFS Unload Utility
<http://www.ibm.com/servers/eserver/zseries/zos/racf/irrhfsu.html>
- ▶ IBM RACF: RACF Downloads and Sample Materials
<http://www-1.ibm.com/servers/eserver/zseries/zos/racf/goodies.html>

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

Symbols

/bin 17
/bin/sh 29, 38
/etc 7, 16
/etc/init.options 29
/etc/log 29
/etc/rc 17, 29
/sbin 17
/usr 17
/usr/sbin/init 29
_BPX_SHAREAS 108
_BPX_UNLIMITED_SPOOL 65
_check_resource_auth_np() 49, 59, 62
_login() 49, 55, 59
_password() 49, 59
_spawn() 55
“a” extended attribute 79
“l” extended attribute 79
“p” extended attribute 58, 79
“s” extended attribute 79

A

AC=1 106
access control list (ACL) 80
access modes 10
Accessor Environment Element (ACEE) 89
ACL 80, 86, 90, 93
 access ACL 87
 directory default ACL 87
 file default ACL 87
AF_INET 21
AF_INET6 21
AF_UNIX 21
AIM 30, 40, 42
ALWAYS 117
APF 79, 106–108
Application Identity Mapping (AIM) 42
As 15
ASSIZEMAX 35, 65
audit attributes 80
auditing flags 115
AUDITOR 80, 110, 115, 118
AUTOUID 42–43

AUTOUID 42–43

B

base ACL entries 80
BASIC 70–71
BPX.CONSOLE 121
BPX.DAEMON 30, 56–57, 61, 69, 121
BPX.DAEMON.HFSCCTL 61, 122
BPX.DEBUG 69, 122
BPX.DEFAULT.USER 34, 36, 39–40, 53, 57, 106, 116
BPX.FILEATTR.APF 69, 108, 122
BPX.FILEATTR.PROGCTL 61, 69, 108, 122
BPX.FILEATTR.SHARELIB 69, 109, 122
BPX.MAINCHECK 69, 71
BPX.NEXT.USER 42, 122
BPX.SAFFASTPATH 81, 122
BPX.SERVER 30, 62–64, 69, 92, 123
BPX.SMF 123
BPX.SRV.surrogated_id 62, 123
BPX.STOR.SWAP 123
BPX.SUPERUSER 64, 66, 123
BPX.UNLIMITED.OUTPUT 123
BPX.WLMSEVER 124
BPX1ACK 126
BPX1SEG 125
BPX1SEU 125
BPX1SGI 125
BPX1SPN 126
BPX1SRG 126
BPX1SRU 126
BPX1SUI 125
BPX1TLS 126
BPX4ACK 126
BPX4SEG 125
BPX4SEU 125
BPX4SGI 125
BPX4SPN 126
BPX4SRG 126
BPX4SRU 126
BPX4SUI 125
BPX4TLS 126
BPXAS 17, 26

- address spaces 29
- BPXOINIT 26–27, 29
 - address space 27
- BPXPRMxx 21
- BPXROOT 56, 58

C

- c89 19
- CATEGORY 99
- chaudit 80, 115, 117
- chgroup 107
- child
 - address space 47
 - process 4
- chlabel 102
- chmod 82, 85, 109, 117
- chown 107, 117
- CHOWN.UNRESTRICTED 66, 107
- clean address space 59
- CLIST 58
- Common Desktop Environment (CDE) 6
- controlled environment 57–58, 60
- controlled library 61
- cpio 16
- CPUTIMEMAX 35, 65
- cron 18, 69
- current connect group 32

D

- DAC 98
- daemon 8, 17
- Data Facility System Managed Storage (DFSMS) 74
- DB2 Tables 120
- dbx 16
- dbx debugger 16
- DEFAULT 117
- default group 32
- DFS 22
- DFSMS 19
- digital certificate 34
- DIRACC 116
- directory list 82
- directory search 82
- DIRSRCH 116
- dirty address space 107
- dirty bit 59
- discretionary access control (DAC) 98, 101

- disjoint security labels 100
- Distributed File System (DFS) 22
- domination 100
- dual identity 32, 53–54
- dubbing 15, 28, 36, 38–39, 47–48, 53, 116

E

- ed 16
- effective GID 49
- effective UID 49, 86, 90
- enable superuser mode 64
- ENHANCED 71
- enhanced program security 69–70
- ENHANCED-WARNING 71
- ENQ 47
- equivalence 100
- exec() 47
- EXECUTE control 70
- extended attributes 69
- external link 25
- external security manager 19

F

- FAILURES 117
- FIELD class 36
- file owner auditing flags 115
- file security packet (FSP) 77
- file system 74
- FILE.GROUPOWNER.SETGID 86, 106
- FILEPROCMAX 35, 65
- fork() 16–17, 46–47
- FSOBJ 116, 118
- FSP 25, 77, 86, 89, 107
- FSSEC 80, 88, 90, 92, 116–117
- FSUM2386 37
- full mode 21

G

- getpwnam() 56
- GID 8
- granularity of privileges 64, 66

H

- hard link 25
- HFS 19, 22–23, 74, 102
- Hierarchical File System (HFS) 102
- HOME 34

I

ICETOOL 41
ICH408I 114
ICH408I messages 20
ICHEINTY 43
ICHRRCDX 116
identity switch 53, 59
IKJEFT01 70
inetd 18, 69
initACEE 33
interprocess communication (IPC) 8
IPC 8, 71–72, 103
 key 72
IPC security packet (ISP) 72
IPCOBJ 116, 118
IRR.PGMSECURITY 70–71
IRRADU00 119
IRRDBU00 41, 94, 119
IRRHFSU 94, 120
ISHELL 24
ishell 16
ISPF 21

J

JOBLIB 58, 79
JWT 19

K

Kerberos ticket 34
kernel 4
kill() 16, 40

L

Language Environment 19
link-edit 106
LINKLIST 79
list-of-groups 54, 81, 91, 93
LNKLST 58
logical file system 6
Lotus Notes 30
LPA 60, 71, 79
LPALST 70
ls 17

M

MAC 98, 101
MAIN 70–71

mandatory access control (MAC) 98
MEMLIMIT 35, 65
message queues 71
minimum mode 21
MIXEDCASE 33
MLACTIVE 101
MLFSOBJ 101, 103
MLIPCOBJ 101, 103
MLS 97, 101
MMAPAREAMAX 35, 65
MOUNT 75–76
mount 19, 116
 security 76
multilevel security (MLS) 97

N

Network File System (NFS) 22
NEVER 117
NFS 21–22, 74
NOPASSWORD 27, 32, 39, 57
NOSECURITY 76
NOSETUID 76
NOUID 36, 39
Novell directory 30

O

octal representation 10, 83
oedit 16
OMVS 26, 75
 address space 26
OMVSGRP 27
OMVSKERN 27
ONLYAT 43
Open Edition 11
Open System 2
OPERATIONS 52, 93
OPERCMDS 30
orphaned ACL 95
owning GID 78, 80, 82, 93
owning UID 78, 80, 82

P

passticket 33, 64
password phrase 33
pax 16
permission bits 81, 83
PFS 7, 21, 23

Physical File System (PFS) 23
PID 19, 27, 47
pidaffinity() 40
pipe 22
PL/I 38
POSIX 2, 12, 38, 47, 102
 POSIX 1003.1 34
 POSIX.0 3
 POSIX.1 3, 48
 POSIX.2 3
prevention of UID sharing 40
PRIVILEGED 53, 90
PROCAT 116
PROCESS 116, 118
process 4, 46
process ID (PID) 19
PROCUSERMAX 35, 65
PROGRAM 35, 58–59
program access to data set (PADS) 70
program control 54, 69, 107
ps 16
pthread_create() 47
pthread_security_np() 49, 55, 59, 62

Q

quiesce 19

R

R_admin 43
R_usermap 34
RACF
 auditor auditing flags 115
 BPX.DEBUG 69
 BPX.FILEATTR 69
 BPX.FILEATTR.APF 69
 BPX.FILEATTR.SHARELIB 69
 BPX.SERVER 69
 BPX.SUPERUSER 66
 UNIXPRIV class 66
 CHOWN.UNRESTRICTED 66
 SUPERUSER.FILESYS 67
 SUPERUSER.PROCESS 67
RACF Remote Sharing Facility (RRSF) 43
RACFICE 119
RACROUTE 43
RDWR 76
READ 76
real GID 49

real UID 49
Redbooks Web site 128
 Contact us xi
Resource Measurement Facility (RMF) 20
RESTRICTED 91
RESTRICTED.FILESYS.ACCESS 91, 94
REXX 17
rlogin 8, 16
rlogind 18, 69
root 9, 17
root file system 23–24, 74

S

SAF interface 19
sanction list 108, 111
saved GID 49
saved UID 49
SECDATA 99
SECLABEL 99, 102
SECLEVEL 99
SECURITY 76
security level 98
sed 16
semaphores 72
server 47
seteuid() 49, 59
setfacl 82, 117
setgid 78–79, 107, 109
setreuid() 49, 59
SETROPS GRPLIST 81
SETROPTS AUDIT 118
SETROPTS LOGOPTIONS 117
SETUID 76
setUID 49, 59
setuid 78–79, 107, 109
setuid() 49, 55, 59
sh_cmd & 16
shared library 109
shared memory 71
shared space 108
SHARED.IDS 41–42
shell
 Bourne Again shell 5
 Bourne shell 5
 C shell 5
 Korn shell 5
 scripts 15
 TC shell 5

SHMEMMAX 35, 65
signals 5
sigqueue() 40
SMF 19
 type 80, 114, 118
socket 22, 38
spawn() 16–17, 46, 49, 59
STEPLIB 58, 79
sticky bit 17, 71, 79, 109
su 49, 55, 59, 64
SUCSESSES 117
superuser 17, 52
 granularity 66
SUPERUSER.FILESYS 67, 91, 93–94
SUPERUSER.FILESYS.CHANGEPERMS 67
SUPERUSER.FILESYS.ACLOVERRIDE 93
SUPERUSER.FILESYS.CHANGEPERMS 82, 88,
109
SUPERUSER.FILESYS.CHOWN, 107
SUPERUSER.FILESYS.MOUNT 66–67, 76
SUPERUSER.FILESYS.PFSCCTL 67
SUPERUSER.FILESYS.QUIESCE 67
SUPERUSER.FILESYS.VREGISTER 67
SUPERUSER.IPC.RMID 67, 72
SUPERUSER.PROCESS 67
SUPERUSER.PROCESS.GETPSENT 67
SUPERUSER.PROCESS.KILL 67, 118
SUPERUSER.PROCESS.PTRACE 68
SUPERUSER.SETPRIORITY 68
SURROGAT 54
surrogate 61–64
symbolic link 25
SYSLOGD 17, 69
SYSMULTI 102
System Display and Search Facility (SDSF) 20
System Management Facility (SMF) 19

T

tar 16
TCB 48
telnet 8, 16
TFS 22, 74
Thread 47
THREADSMAX 35, 65
Tivoli Decision Support 120
trace() 40
TRUSTED 27–29, 53, 75, 90
TSO/E 20

TSO/E MOUNT 24

U

UID 8
umask 84–85
undub 16
UNICS 2
UNIX 95 88
UNIX branding. 12
UNIX System Services 12
 extended attributes 69
 superuser granularity 69
UNIXPRIV 41, 65–66
UNIXPRIV RACF class 66
 SUPERUSER.IPC.RMID 67
 SUPERUSER.PROCESS 67
 SUPERUSER.SETPRIORITY 68
unmount 19, 116
unquiesce. 19
user name 31–32
user security packet (USP) 77
userID 31
USERIDALIASTABLE 32
USP 77, 89–90
USTAR 34
uucpd 69

V

VSAM linear data set 19, 74

W

WHEN(PROGRAM) 58, 60
Workload Manager (WLM) 17
write down 101

X

X.509 V3 34
XPG4 3, 71
X-Windows 5

Z

z/OS Security Server 20
zFS 19, 23, 74, 102

Archived



z/OS UNIX Security Fundamentals



Redpaper

The explanation of the z/OS UNIX security model

This IBM Redpaper introduces the z/OS UNIX security model and implementation to MVS knowledgeable and security-minded users. It does not address in detail all the wealth of specific security features available in z/OS UNIX, but rather the base principles of operation and the mechanisms implementation with setup recommendations.

The use of SAF to achieve superior security

We assume that the user already has a knowledge of the most commonly used IBM Resource Access Control Facility (RACF) setups and commands. However, we do not provide detailed procedures and explanations about the use of these commands.

Professionals' tips and recommendations to achieve superior security

**INTERNATIONAL
TECHNICAL
SUPPORT
ORGANIZATION**

**BUILDING TECHNICAL
INFORMATION BASED ON
PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks