NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DESIGN AND IMPLEMENTATION OF NFS FOR A MULTILEVEL SECURE SYSTEM

by

Kandy Q. Phan

March 2004

Thesis Advisor: Cynthia E. Irvine
Co-Advisor: David J. Shifflett

Approved for public release; distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2004	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Design and Implementation of NFS for a Multilevel Secure System		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Kandy Phan			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release, distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

Popular software for high assurance systems is not readily available. Developers do not want to develop or port applications for secure systems because of the perception that high assurance development is too time consuming, in some cases impossible, and that performance is inadequate. This trend must be stopped by showing that if an intelligent approach to porting software is used, then the development costs will be acceptable.

The network file system (NFS) service, which is a rather complex module that provides widely used functionality for file sharing, has been ported to the XTS-400 to show that a port can be completed in a timely manner and to assess the challenges of development for a multilevel system. Porting starts by analyzing the major requirements of the software and of the target system, and then proceeds to developing an approach for tackling the port.

The hardest part of porting is the learning curve required to understand the target system and the software to be ported. Once this is accomplished, then porting becomes straightforward. Tests demonstrated that remote clients were able to access shared files on the NFS server. The XTS-400 now has the capability to share files through the popular NFS protocol.

14. SUBJECT TERMS Network File System, XTS-400, Multilevel Secure, High Assurance, Porting			**15. NUMBER OF PAGES** 96
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

DESIGN AND IMPLEMENTATION OF NFS FOR A MULTILEVEL SECURE SYSTEM

Kandy Q. Phan
Civilian, Naval Postgraduate School
B.S., University of California, Riverside, 2001

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2004

Author: Kandy Q. Phan

Approved by: Dr. Cynthia E. Irvine
Thesis Advisor

Mr. David J. Shifflett
Co-Advisor

Dr. Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Popular software for high assurance systems is not readily available. Developers do not want to develop or port applications for secure systems because of the perception that high assurance development is too time consuming, in some cases impossible, and that performance is inadequate. This trend must be stopped by showing that if an intelligent approach to porting software is used, then the development costs will be acceptable.

The network file system (NFS) service, which is a rather complex module that provides widely used functionality for file sharing, has been ported to the XTS-400 to show that a port can be completed in a timely manner and to assess the challenges of development for a multilevel system. Porting starts by analyzing the major requirements of the software and of the target system, and then proceeds to developing an approach for tackling the port.

The hardest part of porting is the learning curve required to understand the target system and the software to be ported. Once this is accomplished, then porting becomes straightforward. Tests demonstrated that remote clients were able to access shared files on the NFS server. The XTS-400 now has the capability to share files through the popular NFS protocol.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Dr. Cynthia Irvine and David Shifflett, without whom this would have not been possible. This material is based upon work supported by the National Science Foundation under Grant DUE-020762. I would also like to thank God for hope.

But Jesus beheld them, and said unto them, With men this is impossible, but with God all things are possible.

Matthew 19:26

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Developing software for a multilevel secure (MLS) system, contrary to the general belief that it is difficult, is, in fact, rather manageable. Beginning with an analysis and following up by selecting the right approach makes porting software to high assurance systems quite cost effective. Developing NFS for the XTS-400 will be used as a case study for experimenting with a porting methodology.

Definitions will be provided for MLS systems, the XTS-400, and the NFS protocol. A methodology for porting will be presented and explanations will be given for the necessary tools and common pitfalls encountered when porting software.

The results of the porting experience are summarized and lessons are gleaned from these results. In addition, suggestions for better possible approaches are presented. Also, future extensions to this project are outlined. This experiment concludes that the hardest part of the port is the initial learning curve required to understand the target system and software to be ported. The later phases of development are less problematic.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION/PURPOSE

In today's computer environment, criminal activity is becoming more commonplace because computers are both vulnerable and pervasive. Important activities are done on computers now, and sensitive information that needs protection is being processed through computers. So there is a dire need for secure computer systems. But users are hesitant about the future of secure systems. Users like to have their favorite software packages available. Customers are accustomed to certain software and will not use a system if it is not available. It seems that developing popular software for existing secure systems would solve this problem, but software developers are not doing so because they assume that developing for secure computers is too costly.

It will be shown that valuable software with complicated requirements can be successfully ported to a secure system. This experiment can be used as an example to show effective practices and pitfalls in the porting process. With such an example, perhaps the assumption that developing for secure systems is cost prohibitive will be dispelled, and it will pave a path for other important software to be ported. The value of secure systems will be explained along with the unique challenges for developing software for them. The ported software in this case study will be the network file system (NFS) service that is popularly used among commodity hardware for file sharing.

B. MLS: MULTILEVEL SECURITY

Multilevel security (MLS) is an important concept that needs to be widely deployed. MLS provides for files to be classified at different confidentiality and integrity levels and to only allow access to a file if the user has sufficient authorization. Common levels of classification are *Unclassified*, *Classified*, *Secret*, and *Top Secret*. A more in-depth definition of multilevel security controls is [SCHELL74]:

> those controls needed to process several levels of classified material from unclassified through compartmented top secret in a multi-process multi-user computer system with simultaneous access to the system by users with differing levels of clearances.

MLS is required in many areas, such as the government sector, that processes data related to national security and for the business sector where information must be protected to ensure privacy. In the digital world, components that must enforce MLS policies include operating systems, file systems, and databases.

Without an MLS system, a user would require three different computers to process information at three different classification levels. This would be quite unwieldy. It would be much more convenient if all this different information could be available on the same machine. Also having information on separate computers is extremely inefficient and requires more equipment than necessary. This separate computer model is burdensome in a facility that requires information sharing. In 1972, the Computer Security Panel estimated that separate systems require the additional cost of $100 million per year for the Air Force [SCHELL74]. But if all the information at different levels is on the same computer, great care must be taken to ensure that information is not inadvertently leaked to another classification level.

The main objective of an MLS system is the enforcement of a mandatory access control policy. This is based on Bell and LaPadula's (BLP) formal security model [BELL73]. BLP is used to model the policy in such a way that it can be analysed. Also, the mathematical formulation allows it to be mapped to a technical implementation. The simple security property of this model prohibits “data at a higher level from being read by a process at a lower level" [PERRINE]. The other rule, the star (*) property, "prohibits a process at a higher level from writing to a lower level data" [PERRINE].

Related to the concept of MLS, a *Security Kernel* is normally mentioned to make sure that MLS is properly enforced. The kernel enforces the security policy and constrains the operating system layer. The *Security Kernel* is small and only provides very basic functions. It must be small so that it can be verified;

verification ensures that the security policy is properly enforced. Then an operating system layer, which provides functionality but does not need to be secure because it does not enforce policy, can be built on top of the small kernel. This was done for several systems such as KSOS [PERRINE], GEMSOS [GEM], and STOP [MANU]. The *Security Kernel* is a necessary ingredient for high assurance systems.

Some systems that have implemented MLS policies. The Kernelized Secure Operating System (KSOS) is one such system. KSOS has three levels of security [PERRINE]. The security kernel of KSOS was designed and modeled after a finite-state machine and system calls are defined in terms of state transitions [PERRINE]. The security kernel is verified by defining an initial known secure state and then checking to see that all possible state transitions (system calls) led to a secure state [PERRINE]. KSOS has only 32 kernel calls and thus is manageable. KSOS was used by the Navy and Air Force to provide "multi-level secure application gateways on very secure DoD networks" [PERRINE]. The problem with this system and other high assurance systems, is that little software is available for them.

C. MYSEA: MONTEREY SECURITY ENCHANCED ARCHITECTURE

The Monterey Security Enhanced Architecture (MYSEA) project goal is to provide a high assurance architecture that "supports mandatory enforcement of confidentiality and integrity policies in hostile environments" [IRV04]. It achieves this by using commodity products that are readily available and affordable. The other advantage of MYSEA's use of commodity products is that users are already familiar with these products, which leads to a higher likelihood of acceptance for high assurance systems.

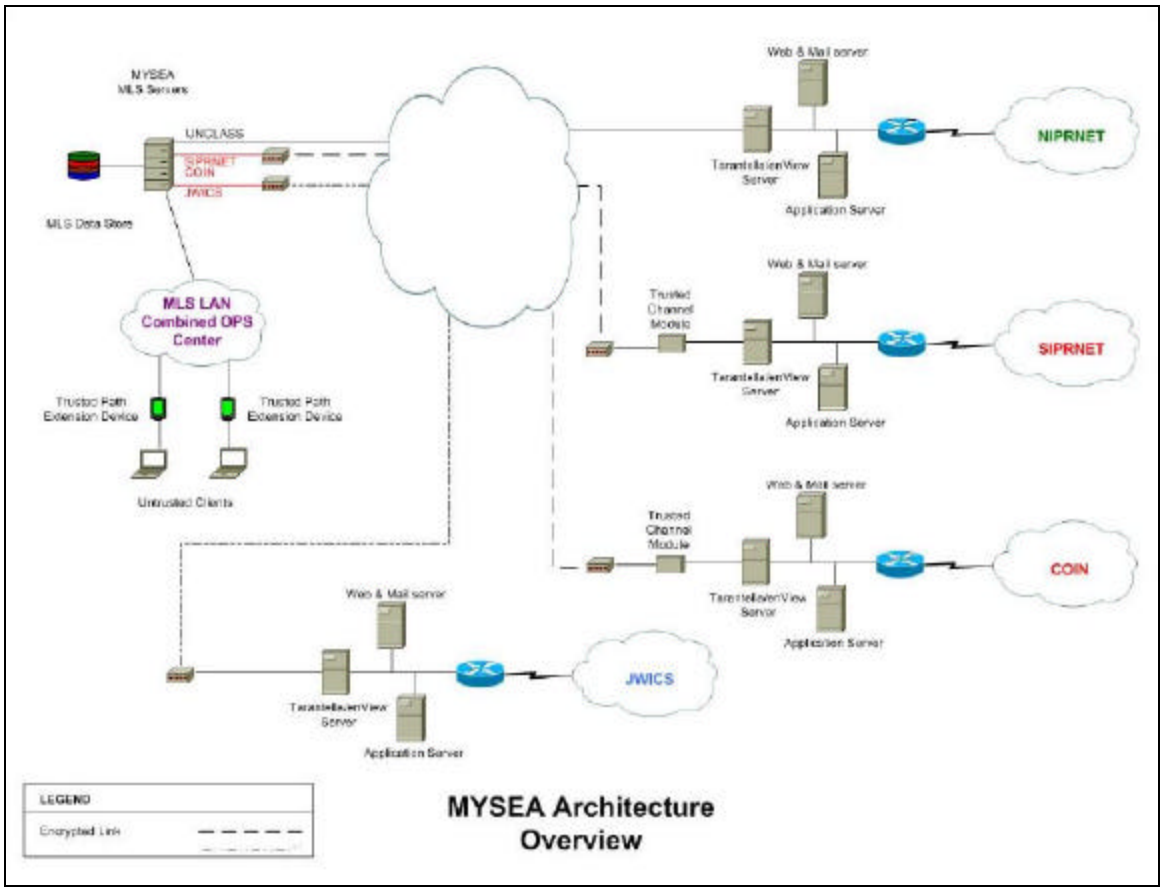

Figure 1. Monterey Security Enhanced Architecture, From [IRV04]

At the heart of MYSEA is the XTS-400, which acts as a secure server for hosting applications. The XTS-400 was chosen for its high assurance capabilities. Properties of the XTS-400 that make it a good fit for the MYSEA project include: high assurance multilevel platform, remote trusted path, Linux binary compatibles, and correct security policy enforcement [IRV04]. These factors also make the XTS-400 a good candidate for use as a target system for a port.

The untrusted clients in the MYSEA architecture require a file system to function properly, but the file system must be on the MLS server, the locus of the security policy enforcement. So the untrusted clients need to use a remote, bootable file system that comes from a trusted server. NFS provides this

capability. By creating an NFS port for this system, the MYSEA project will be even more attractive and this will help to ensure its success as a trailblazer for useful secure systems.

D. SUMMARY

MLS systems are needed for the secure processing of sensitive information [TP85]. This overview of MLS shows what is needed and what is different for developing these types of systems. There has been some prior work on secure systems, but it was not successful due to a lack of important software. Chapter II provides information on NFS, the target system XTS-400, and why porting software to different hardware/software platforms is difficult. Chapter III shows how to approach the port, what type of environment the XTS-400 provides for the program and the necessary tools for porting. Chapter IV illustrates how to test the port to see if it is successful. Chapter V describes the lessons learned from implementing the port and reflects on how the NFS port could have been done better.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. STOP: SECURE TRUSTED OPERATING PROGRAM

1. Overview

The XTS-400, which runs on Intel-based hardware, is the successor to the XTS-300. The XTS-300 is one of a few systems that was evaluated at the Class B3 level. Under the Trusted Computer System Evaluation Criteria, systems evaluated at Class B3 level provides a high level of assurance of correct policy enforcement and functionality [TP85]. Requirements for Class B3 systems include [WANG99]:

> enforcement of mandatory and discretionary access control polices, a trusted path to ensure a reliable TCB-to-user communication connection, and an alarm mechanism to detect the accumulation of events that indicate an imminent violation of the security policy.

The trusted computing base (TCB) of a system is the set of hardware, firmware, and software that is responsible for the enforcement of a security policy [TP85]. The TCB needs to enforce the principle of least privilege, which only gives users the necessary functionality to perform their task [SALT75]. The TCB also needs to adhere to a modular, abstract, and data-hiding design to reduce complexity.

STOP is the operating system for the XTS-400 system. STOP, which is a multilevel secure operating system, provides virtual memory, multiprocessing, and "uses hardware protection level mechanism in conjunction with software mechanism for protection" [WANG99]. The formal security model implemented by STOP is the Bell and LaPadula model [BELL73]. This model outlines a confidentiality policy for both mandatory and discretionary access control. The mandatory component of the Bell and LaPadula model states that users can read information at their current level and lower; the users can write information at their current level and higher. The integrity model used by STOP is the Biba model [BIBA]. The Biba model specifies that users can only read data that is of equal or higher integrity than their level and can only write data at their current level and lower.

2. Ring Architecture

A ring architecture divides a system into layers so that the most privileged operations are protected in the innermost layer and the least privileged operations are performed in the outer layers. The outer layers depend on the inner layer for trusted functionality, and in order for a request from the outermost layer to reach the innermost layer, that request must be cascaded through all the intermediate layers. Due to the clear distinction between layers and the simplicity of this model, the ring architecture is found in several operating systems that have security as a design objective. This was first implemented in Multics [MULT] and later in GEMSOS [GEM]. STOP uses four rings.

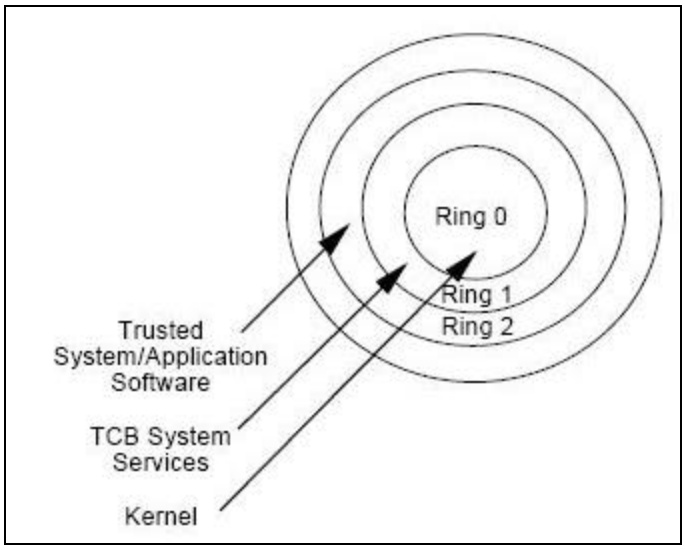

Figure 2. XTS-400 Ring Architecture, From [WANG99]

Here the rings of the XTS-400 are described [WANG99]. The most trusted layer, Ring 0, is the *Security Kernel*. This layer provides basic operating system services and enforces system security. These services include resource management, process scheduling, interrupt and trap handling, auditing and the enforcement of a mandatory security policy, such as a set of rules for enforcing system security and system integrity policies [MANU].

Ring 1 contains the *Target Of Evaluation* security functions (TSF) *System Services*, which provide Ring 2 with general trusted services, such as a hierarchical file system, user I/O, and discretionary access control [MANU].

Ring 2 contains the *Operating System Services*. This layer provides an environment on the XTS to support the execution of Linux-based application programs [MANU]. It also provides trusted operating services to both trusted and untrusted application software.

Untrusted Application Software and *Trusted Software* are located in Ring 3. *Trusted Software* includes additional security services outside the *Security Kernel*. The untrusted software is constrained by the security policy of the TSF.

B. NFS: THE NETWORK FILE SYSTEM PROTOCOL

1. Need

In today's information-based world, the ability to share data is crucial. Researchers in different departments must collaborate on projects, and important business documents must be edited and reviewed by partners in remote locations. Computer networks have made these capabilities a reality, but a slew of supporting software is also needed. A necessary element to make file-sharing efficient is a file system that can be accessed remotely. An early method of file-sharing required users to access a central machine remotely where the shared files were located. Unfortunately, these central machines became a bottleneck because the numerous requests by clients overloaded the system. This problem spurred a demand for an efficient method to share files across machines.

2. Requirements for a Remote Distributed File System

For a remote file system to be considered adequate, it should meet some basic requirements. An important requirement for a remote distributed file system is support for a wide array of platforms and operating systems. This means that users are not limited to using only certain systems. The ability to share files among heterogeneous systems is very useful; thus, the network can consist of a combination of different systems, and users are still able to share files. So it is necessary to take steps during the design process to keep interoperability in mind.

Secondly, access transparency is important so that existing programs do not need modification to work properly with the remote file system [COUL]; this helps to allow easy adaptation of remote file system services. Failure

transparency is also a good attribute to have since in a network environment there are multiple sources where failure can occur.

Scalability is another important need so that the design can support an environment that requires files to be shared among a large user base. Related to scalability, is the requirement for a shared database of necessary information, such as usernames, passwords, and available file systems, that can be easily updated and accessed. Where and how this database is maintained needs careful consideration. Lastly, the use of a remote file system should not have noticeable performance issues (i.e. it should not take longer than four seconds to get a response from the server).

3. Design

A simple file system model was created to solve the file-sharing problem. This model entailed servers exporting their file systems to a group of clients, and the clients importing these remote file systems and presenting them to the user as a local file system.

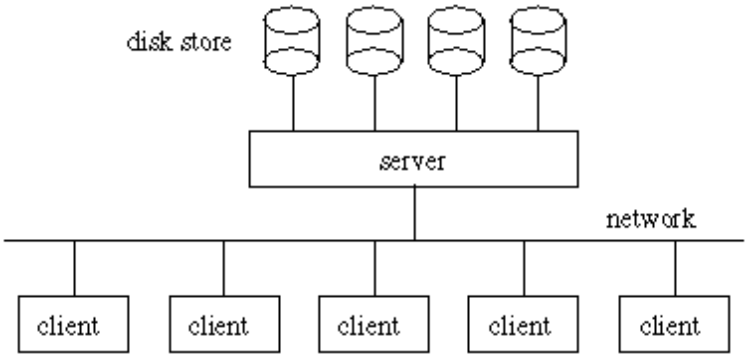

Figure 3. The Division of NFS between Client and Server, From [MCKU]

One of the remote file system protocols that is based on this model is the Network File System (NFS) created by Sun Microsystems [MCKU]. With this model, NFS provides the capability to share file systems over a network of computers. A distinct feature of the NFS design is its high level abstractions, which makes it interoperable with many different local file systems.

NFS is specified in Request for Comments (RFC) 1094 [SPEC89]. An RFC is a standard protocol that has been finalized by the Internet community. As

a public specification, NFS enjoys the benefits of being a standard and has implementations on a variety of machines. Another factor that has helped NFS gain wide support is its reliance on RPC, which is available on a wide base of systems.

The NFS service is comprised of three main components: the Remote Procedure Call (RPC) portmapper, the mount daemon, and the NFS daemon. The portmapper is responsible for registering RPC services and provides program lookup information. Mappings between a port number and a program are handled by the portmapper. When an RPC daemon starts up, it tells the portmapper which port it is using and the services (service name and version number) it will be handling. A client wishing to access an RPC service will first contact the portmapper to find out which port to use.

The mount daemon is responsible for the initial request for a remote directory share. Its other duties include validating the client's credentials and checking for proper access permissions. A successful reply from the mount daemon will include a file descriptor, a handle that the client will use in future requests for operations on that file [MCKU]. Lastly, there is the NFS daemon, which handles the basic remote file operations such as read and write.

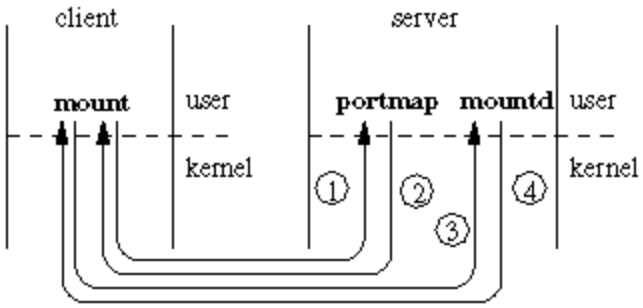

Figure 4. Daemon interaction when a remote file system is mounted, From [MCKU]

NFS is made to be high level and modular, so that it does not have strict requirements on the lower layer. This allows NFS to run above a variety of different file systems, and the underlying file system may be changed to meet different operational requirements.

NFS attempts to be as stateless as possible; servers do not need to store client-state information to function properly. This stateless design has advantages in failure recovery. For a stateless server, if the server crashed and was brought up again, the client would just need to resend its request. On the other hand, a stateful-server crash means that the client must rebuild the server state to know if the server received the client's last request. Only then will the client know if it has to resend its request.

4. Details

NFS relies on the Remote Procedure Call (RPC) model [SPEC89]. The RPC protocol provides a procedural interface to services, allowing a client to request a server's operation through a procedure call. The RPC model views a server program, such as NFS, as a set of procedures. The combination of host address, program number, and procedure number uniquely identifies one remote procedure.

NFS provides UNIX semantics for its procedures, so viewing it from a UNIX perspective is helpful. Most of the UNIX commands for files are supported. Access control follows the UNIX model, but the actual permission check is done at the lower file system layer and can therefore be implementation dependent.

NFS is a stateless protocol, but the objects that NFS is required to handle, files and directories, are stateful objects. To handle these objects properly, NFS needs to operate in such a way that the attributes of the objects stay consistent. NFS does this by requiring all of its important procedures to be idempotent, which means "an operation can be performed repeatedly with the same effect as if it had been performed exactly once" [COUL].

NFS can run over TCP, which provides a robust, connection-oriented transport service, or UDP, which provides a minimal datagram delivery service. Running NFS over TCP gives more reliability to NFS but affects performance by requiring the establishment and maintenance of connections. Since NFS is designed to safely recover from failure, the smaller footprint and faster service provided by UDP is sufficient.

Since UDP does not guarantee reliable packet transmission, mechanisms are needed on the layer above it (the RPC level) to provide more reliable packet transmission. The UDP version does this by sending a datagram and setting a timeout value based on the expected time for a reply. When the timeout expires, the datagram is retransmitted.

However, this retransmission can cause problems with the nonidempotent RPC operations because they are unnecessarily repeated. To help resolve this problem, an understanding of round-trip timeout (RTT) is needed. A larger value for the RTT will help avoid the retransmission problem because it allows more time for slow reply packets to reach their destination before an unnecessary retransmission is triggered.

On the client side of NFS, the issue of dealing with failures must be addressed. There are two approaches for dealing with this problem. There is the *hard* mount method in which the client will continue sending its requests to the server forever. The second way is the *soft* mount method, in which after a set timeout period, the procedures will return with a transient error. The problem with the second method is that transient errors are not expected from the file system since they are usually considered to be local [SPEC89]. So applications not expecting transient errors can incorrectly interpret them as a critical failure and exit early. Another issue concerns how long the timeout values should be. Setting it too short can return false positive errors, and setting it long can cause the application to hang unnecessarily long when serious connectivity problems occur. A middle ground option is to have an interruptible mount. This method acts as a *hard* mount, but it can also receive signals [SPEC89]. On the arrival of a signal, the blocked system call will return with a transient error. This allows interactive programs that are waiting on an NFS procedure to return to be aborted.

NFS meets most of the requirements for a remote file system, but it has shortcomings in certain areas. NFS was originally designed to have each server support around 5 to 10 clients, so its scalability is limited [COUL]. NFS also does

not have a distributed system to share its database of users and passwords; each client and server needs its configuration files to be individually changed.

C. THE PORTING PROBLEM

Porting software from one system to another can be difficult. The porter must look for elements in the software that are dependent on the architectural features of the hardware. The reasons these software pieces exist can range from bad programming practices (not developing with portability in mind) to the programming language lacking support for expressing an aim in a portable fashion [PEMB]. In some cases, finding and fixing these trouble spots is quite trivial, but for a code section that implements complicated operations or may have a complex chain of dependencies on other code modules, the porter can be faced with quite a quandary when trying to figure out how to port it.

Data types cause numerous porting problems. This can be seen in the range of data types: the maximum and minimum values a type can take on. Whether a data type is signed or not will affect this range. The range problem can be seen if an *int* is expected to be 32-bits on the source system, but on the destination system *int* has only 16-bits; the program's calculations can unexpectedly overflow and lead to unreliable results.

Floating point data types present more complicated problems, stemming from its properties and how it is represented on the machine. According to Pemberton [PEMB], floating point numbers are represented as a "fraction *f* of some fixed number digits, in some base *b* (usually 2, but 16 is not unknown) and an exponent *e*, so that a given number is represented as *f* x *be*". The following must be taken into consideration for floating point numbers: the base used, the accuracy available, the maximum and minimum values, and epsilon [PEMB]. Epsilon is the smallest value that can be added to 1.0 to give a different value and is usually dependent on the rounding mode used by the arithmetic [PEMB]. This is important when comparisons are made between floating point values and if there is a requirement regarding the level of precision used.

Other data type issues include how conversion between different types takes place and the properties of pointers. Some systems allow certain sorts of pointer arithmetic that are not supported on other systems. These problems usually result in dereferencing null pointers or pointers pointing to strange locations.

At the lower level, details about how data is stored can affect porting. For byte ordering, there is the possibility of little-endian or big-endian. Systems can have different definitions for the length of a word. For example, a word could be represented by either two or four bytes. Data alignment is another architecture dependent attribute. Programming tricks that rely on how variable size is affected by data alignment will break on systems that align differently than what was assumed for the original system.

If assembly language, which is not easily portable by nature, is used in the program, its port could entail a painful one-to-one instruction conversion to the target system. Problems involved in the assembly language porting process include different semantics for instructions, such as byte swapping, and no equivalent instruction on the target system. In the case of porting a *Complex Instruction Set Computer* (CISC) based assembly program to a *Reduced Instruction Set Computer* (RISC) system, not only is there lots of dissimilarity between the instruction sets, but also an entirely different paradigm of programming is required. The assembly language world is rich with programming tricks. These tricks are applied at the bit level and make assumptions about how overflow is handled, how flags are set, and how bits are shifted. All of these can lead to disastrous results on a system that does not support them or implements them differently, if porting is done by rote. Instead a careful analysis of the intended program semantics is required. When this is completed, it is often less costly to rewrite the software from scratch rather than to try to port it.

Assembly language is tightly coupled with the processor of a system. With the processor in view, a whole slew of factors become apparent. This

includes how pipelining affects the program flow, what sort of branch prediction is used, and quirks that are unique to the processor. There are also hardware-timing issues based on granularity of a time quantum for different processors.

Although programming languages are standardized, their implementation can vary. For instance, for the Windows environment for Visual C++ 6.0, defining a counter variable for a *for* loop gives that variable scope outside of the *for* loop. On the other hand, for the UNIX environment for the GCC compiler, the counter variable only has local scope inside the *for* loop. Compiler optimizations can also cause a program to behave differently. Programs can be reliant on compiler-specific features that are only available on a specific system. A development environment might also have taken up namespace that is needed by the program and thus causes a conflict. Lastly, how the compiler stores data in memory, how it affects the memory stack, and whether it grows the memory stack up or down needs to be known in order to port programs.

As can be seen, many issues can factor into the porting process. Each program has its own unique challenges that must be understood and cannot just be automated to have an equivalent in a new system. Having to understand and deal with all these possible roadblocks leads one to the conclusion that porting is indeed a challenging problem.

D. NFS PORTING PROBLEM

NFS is a multi-component software application. Because of the inter-dependency of its subcomponents, NFS is harder to port than single component software. The different layers cannot necessarily be tested thoroughly individually. A method will need to be devised that can test the components together as a whole, as well as allow periodic checks to gauge that each component requirement is being met. Besides being multi-component, NFS is also heavily dependent on the local file system mechanics and the system calls behavior. The dependencies of NFS are shown in Figure 5.

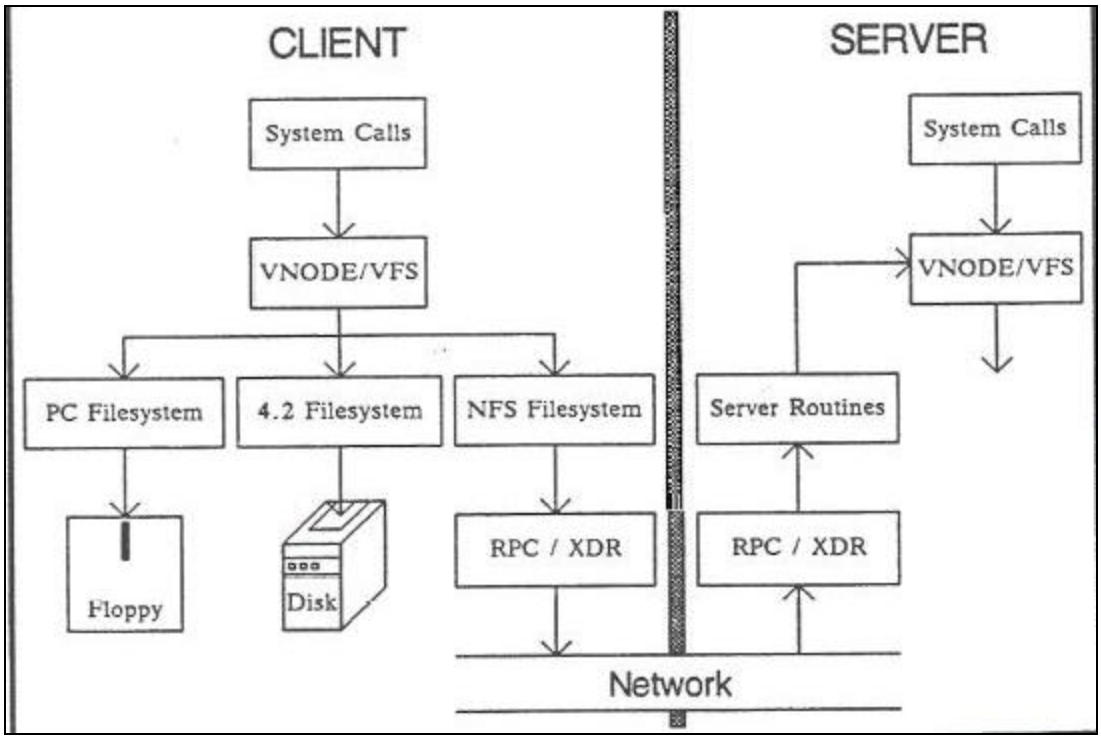

Figure 5. NFS dependency chart, From [SAN85]

Another obstacle for porting NFS is that it does not have MLS capabilities built into it. In fact, it cannot even distinguish between different user-ids unless it is used in combination with the yptools[SAN85]. A decision will need to be made on how to remedy this problem.

In order for NFS to be considered complete, it must support the following operations[SAN85]:

Filesystem Operations:

mount(varies) System call to mount filesystem

mount_root() Mount filesystem as root

VFS Operations:

unmount(vfs) Unmount filesystem

root(vfs) returns (vnode) Return the vnode of the filesystem root

statfs(vfs) returns (fsstatbuf) Return filesystem statitics

Vnode Operations:

open(vnode, flags) Mark a file open

close(vnode, flags) Mark a file closed

rdwr(vnode, uio, rwflag, flags) Read or write a file

ioctl(vnode, cmd, data, rwflag) Do I/O control operation

select(vnode, rwflags) Do select

getattr(vnode) returns (attr) Return file attributes

settattr(vnode, attr) Set file attributes

access(vnode, mode) Check access premission

lookup(dvnode, name) returns (vnode) Lookup file name in a directory

create(dvnode, name, attr, excl, mode) returns (vnode) Create a file

remove(dvnode, name) Remove a file name from a directory

link(vnode, todvnode, toname) Link to a file

rename(dvnode, name, todvnode, toname) Rename a file

mkdir(dvnode, name, attr) returns (dvnode) Create a directory

rmdir(dvnode, name) Remove a directory

readdir(dvnode) return (entries) Read directory entries

symlink(dvnode, name, attr, to_name) Create a symbolic link

readlink(vp) returns (data) Read the value of a symbolic link

fsync(vnode) Flush dirty blocks of a file

inactive(vnode) Mark vnode inactive and do cleanup

bmap(vnode, blk) returns(devnode, mappedblk) Map a block number

strategy(bp) Read and write filesystem blocks

bread(vnode, blockno) returns(buf) Read a block

brelse(vnode, buf) Release a block buffer

III. PROPOSED NFS DESIGN IMPLEMENTATION

A. PORTING APPROACH

There are many possible porting approaches, each with unique advantages and disadvantages. Depending on the type of application being ported and the type of environment available for development, one approach will prove to be more effective than the others. This section will outline three common approaches to porting software.

The first approach is to modify the existing libraries. This is done at a rather low level and is required when the application relies on the existing libraries, but the libraries behave in a manner different from what is expected by the software. The ported application, in this case, is very modular in design and does not have system-specific dependency at its own code level. Depending on how these modified libraries are placed in the system, other existing applications might fail to run because they require the original libraries. Fortunately, there are ways to specify that the ported application will use the new, modified libraries, which existing applications will not use. This is via declarations of path environment variables. If this is the case, then two copies of the libraries will be on the system, thus this method consumes extra hard drive space. Library files can total up to 80 MB, so depending on a system's hard drive capacity this can be an issue. Since the application code is not modified, this method enjoys the advantage of not needing any changes when a newer version of the software is needed. A newer version of the application can be moved onto the target system, and the modified libraries are already in place for its use.

The second approach, which is a variation of the first method, is to create a new library that provides the functionalities that are lacking in the existing libraries but are needed by the ported application. The link order becomes an important concern for this method because the new library needs to be linked ahead of the default libraries to remove incompatibilities, so that the application can compile successfully. This method enjoys many of the benefits of the first

approach and has the added advantage of not requiring management of a separate, redundant library specifically for the port. This approach, however, needs the application to be recompiled with the correct link order. The porter must have a thorough understanding of how the system loads binaries and how the system interacts with libraries in order to use this approach.

The third approach is to modify the application. This is required when the ported application uses a capability that is not available on the target system. An example of this is that it is possible to *select* on sockets in UNIX systems, but this this function is not available on Windows systems. The advantage to this method is that the libraries do not need to be changed. The disadvantage of this method is that with each new version of the application, these same changes will need to be reapplied.

Depending on the application being ported, it is possible that a combination of all three approaches will be needed. For instance, a porter can modify an existing library to handle a system call correctly, add required functions that were not available in the existing libraries to a new library, and change sections of the ported software that used a programming paradigm that is not available on the target system. On the other hand, when any method will work for an implementation, a decision must be made regarding which method to use so that consistency in the program will be maintained.

The first method was chosen for the NFS port because it is believed to be the quickest method to get the application ported. This decision was made from a quick analysis of the NFS source code. The conclusion from the analysis is that the NFS-relevant code was quite modular and would be able to work without any changes, but there were dependency problems on library functions, which behaved differently on the XTS system. These library functions would need to be changed. Since only existing library functions needed modification, and figuring out the linking environment of the XTS system so that method two could be used would be more time consuming, the first approach was chosen for the NFS port.

B. STOP PROGRAMMING ENVIRONMENT

The XTS-400 target system is missing important porting tools such as *gdb*. Due to the restrictive nature of this XTS-400 system, it was more efficient to develop on a more flexible development system that was made to be a cross-platform development environment for the XTS-400 system. Luckily the XTS-400 Linux environment is based on the standard Redhat 8.0 distribution [RED], so the same development tools and libraries can be installed effortlessly on an easier to use system. With its more sophisticated development tools, this system can be used to test porting concepts and to compile binaries quickly. These can then be used on the target machine.

The most glaring difference between the XTS-400 Linux environment and that of Redhat is that it uses a 2.2.x kernel rather than the default 2.4.x kernel that comes with Redhat 8.0. This means that differences between these two kernels must be taken into account. Of course, XTS-specific applications and services are not available, but the NFS software is specifically designed to avoid being dependent on them.

The XTS-400 does not provide all UNIX functionality, and it also interprets some conventions differently. These differences are what the porting effort must address. For example, there is no *UID* 0 in the XTS-400. Because of this, the following system calls are affected [MANU]: *chmod*, *chown*, *execl*, *excel*, *exclp*, *execve*, *exevp*, *kill*, *link*, *mknod*, *setgid*, *setuid*, *msgctl*, *semctl*, *shmctl*.

The *real-user-id* and *real-group-id* of a process cannot be modified by Linux system calls. The *set-user-id* and *set-group-id* cannot be changed by Linux system calls, thus *chmod* and *exec* are affected [MANU]. Due to a different handling of date access time (*st_atime*), the following commands are affected [MANU]: *fstat* and *stat*. Mandatory file/record locking is not supported, so the following commands are affected [MANU]: *chmod*, *open*, *read*, and *write*.

At the lower system call level, the following Linux system calls are not available [MANU]:

> acct, adjtimex, bdflush, capget, capset, chroot, clone, create_module, delete_module, fstat, fstatfs, get_kernel_syms, idle, init_module, ioperm, iopl, lstat, mlock, mlockall, modify_ldt, mount, munlook, munlock, munlookall, nfsservctl, old_readdir, old_select, oldstat, oldumount, olduname, prctl, pread, pwrite, query_module, quotactl, reboot, sched_get_priority_max, sched_getpriority_min, sched_getparam, sched_getscheduler, sched_rr_get_interval, sched_setparam, sched_setscheduler, sched_yield, sendfile, setdomainname, setgroups, sethostname, settimeofday, stime, swapoff, swapon, sync, sysctl, sysfs, sysinfo, syslog, umount, uselib, ustat, vhangup, vm86, vm86old.

C. DEVELOPMENT TOOLS

Porting is a complicated process and requires familiarity with a handbag of tools for success. This section will go over the necessary tools that a porter will need along with some hints on how use them properly for porting.

gdb, a debugger for the C programming language, is a crucial tool in the porting process. It can be used to get a quick idea of how a program works and the sequence of functions called by the program. To do this, the porter needs to make breakpoints at the necessary location in the code. Printing values for variables and arguments and finding out how function pointers resolve is a powerful debugging technique that is indispensable for porting. However, *gdb* does run into problems for certain programming styles. For instance, if there is a function within an *if* statement, a step command will not properly go into that function. One can unravel the *if* statement into multiple statements in order for the step command to work, but this unraveling needs to be done very carefully. The semantics of original statements should be fully understood before attempting unraveling.

make, a compiling management tool, and *gcc*, a C language compiler, are mandatory development tools. Using *make* and *gcc* can reveal many errors at compile time caused by source code change. In order to use *gdb*, the source needs to be compiled with debugging flags. The other advantage of *make* is that

it only compiles the files that are affected by any of the changes that were made. This saves a lot of time in the edit-compile-debug cycle.

The X-Windows system of the XTS-400 is a great productivity enhancer. The default console only allows one process to run in the foreground at a time. However, in the X-Windows environment, running multiple xterm terminals with each xterm acting as a separate console is possible. This enables multiple processes to run in their own virtual terminals, which allows multiple server programs to run at the same time in their own foreground terminal. Thus, the porter will be able to see all errors message generated by the different server programs simultaneously.

strace, a tool that tracks the system calls made by a program, is an invaluable tool in the porting process used to track down where a program is crashing. First, the program is run to see if there are any errors. If this is the case, then a quick run of *strace* can possibly show where the problem is in the form of what functions failed and what the function arguments are. It is very likely that *strace* will scroll the terminal too fast. To remedy this problem, the *script* command can be used to capture the output session into a file.

Source code analysis is essential to the success and timeliness of porting software. But the regular format that source is in is not the most conducive for analysis purposes. To help in this area, a tool will be used that turns a source code tree into cross referencing HTML pages that can be read through a web browser. This tool is called *global*. The weakness of this tool is that it did not provide the source code line number in its display, which is a very handy feature that can be used when a program is executing in *gdb* to find its equivalent place in the source code. *lxr* is a cross referencing formatting tool that does display line numbers, but its weakness is that its display is not as appealing as the one provided by *global*.

The source code for STOP was also very useful in the porting process. When a system call is used and returns an error, the system source code can be quickly checked to see if it has been implemented properly. Although

documentation can and should take care of this listing of functionality, it is sometimes not as thorough as it should be. For instance, the incorrect handling of *ioctl* with the argument *SOCGIFFLAGS* is not documented. Having the system source code available will mitigate the problem of deficiency in documentation.

One of the oldest programming tricks for debugging and the first one that a programmer learns is to output important values to the screen through the use of the *printf* function. This technique is very useful. When the porter is stuck, it will not hurt to start using *printf* to output values in various places in the code to hopefully get an idea of how to proceed.

IV. TESTING THE NFS DESIGN

Testing was divided into two stages: component testing and integration testing. The component testing phase was to see if the basic functionality worked, while the integration testing phase is more thorough and demanding to see if the port is complete. Each will be described in the following sections.

A. COMPONENT TESTING

The server is comprised of three components: portmapper, *mountd* daemon, and *nfsd* daemon.

This stage of testing involves testing each of the server components separately to see if they individually work. Each test can first be executed locally, and then with a remote client issuing the test commands.

To test the portmapper, a user program is used to connect to the portmapper and to make requests to the portmapper to see if the portmapper is functioning properly. This test program is called "rpcinfo". A successful result from "rpcinfo" will show that the portmapper has registered itself as an available RPC service.

To test the *mountd* daemon, a user program will be used to contact the mountd daemon and check it to see if a mount request will return with a valid result. This program is called "mount". The "rpcinfo" command will also be used to see if *mountd* has properly registered itself with the portmapper. Also netstat can be used at this time to see if the *mountd* specific ports are open and that the related network connections are established.

To test the *nfsd* daemon, a user will test to see if he can read and write files. Testing of directory creation and deletion will also be conducted. Users can use the commands such as "ls", "touch", "mkdir", and "vi" to do this. These commands will use as arguments the name of files and directories exported by an *nfsd* daemon. Success is indicated by the successful completion of those commands. The "rpcinfo" command will also be used to see if *nfsd* has properly registered itself with the portmapper.

Table 1. Portmapper Test Results

Test Description	Result
"rpcinfo" properly connects with portmapper	yes
show that *mountd* is registered	yes
show that *nfsd* is registered	yes

Table 2. Mountd Test Results

Test Description	Result
"rpcinfo" shows that *mountd* is registered	yes
"mount" command returns sucessfully	yes
"netstat" shows that proper network connection has been made	yes

Table 3. Nfsd Test Results

Test Description	Results
"rpcinfo" shows that *nfsd* is registered	yes
listing the content of a mounted directory	yes
open and read a file	yes

B. INTEGRATION TESTING

The final stage of testing is conducted to determine if clients can mount the NFS directories remotely rather than locally. This testing is more thorough to show that a complete implementation of NFS has been provided. Clients will attempt to modify the files on the server, and a check will be made to see if other

clients can see the changes made. The tests include: adding a file, deleting a file, removing a directory, adding a directory, and changing the contents of a file. There will also be tests to see if the security measures of the implementation work: have a client with no access mount a directory, have a client with read-only access write files, and have a client mount directory A when only directory B is available.

Table 4. Integration Testing

Test Description	Result
Redhat 8.0 client can mount	yes
Slackware 7.0 client can mount	yes
create a directory	yes
delete directory	yes
read a file	yes
write a file	yes
create a file	yes
delete a file	yes
mount from a client not allowed in the configuration file	no
Access a file below user level	yes
Access a file at user level	yes
Access a file above user level	no
read-only client write a file	no

THIS PAGE INTENTIONALLY LEFT BLANK

V. LESSONS LEARNED

A. LESSONS LEARNED

This section will describe some of the problems that were encountered during the process of porting the NFS server.

An issue that increased the complexity for understanding how the program works is the interaction between forked processes that run in the background, logging facilities, and the permission set for log files. These type of actions must be turned off so that the core functionality of the program can be focused on. Fortunately, *portmap* is made so that its daemon mode can be turned off by running it with the '-d' argument. The *mountd* process also comes with an option that allows it to be run in the foreground. To activate this mode, run the *mountd* binary with the '-F' flag. Another advantage of running a process in the foreground is that it is conducive to debugging. This is because daemon processes usually follow the model of forking and then exiting, and when *exit* is called, the debugging session ends.

There was the problem of *ioctl* not working with sockets and the argument *SIOCGIFCONF*. *ioctl* is a generic function that is used to set and get device information. A look through the STOP source code shows that it is not implemented for sockets. A way to get around having to use *ioctl* is to manually provide the information that it would return. One possible solution is to have a stub function in place of the function that calls *ioctl* to just set the *sockaddr_in* argument to the local interface and then return it. This is an inelegant way of fixing this problem, but it works and is effective. This method was possible because the intended use of this function is understood. A more appropriate method would be to actually implement the *ioctl* to work with these arguments.

A further study of the XTS-400 development environment shows that the *SIOCGIFCONF* was actually defined, although not usable, through the header file "xts/devices/socket.h". This is explained in the *man* pages of *ioctl_list*. The *SOCGIFFLAGS* argument to *ioctl*

also does not work with sockets. The purpose of this command is to retrieve the flags that are set for a socket. Returned information includes whether the interface is currently up or down.

B. METHODOLOGY OF PORTING

There are many lessons that can be gleaned from the experience of attempting this port. Looking back at the process of porting NFS to the XTS-400 with the advantage of hindsight, it becomes evident that a better approach could have been taken. This section serves as an outline of the steps that were done that were good, along with other approaches that would have made the porting approach easier.

First, a general understanding of the ported program is needed. Reading the general information on the topic can facilitate this. In this case, reading the relevant RFC [SPEC89] and some general OS books [MCKU] that have a section on NFS would be helpful. Some online sites with general information are also useful.

Next, an intimate knowledge of how to use the software should be acquired. So a configuration needs to be set up so NFS can be tested. The user should note the kind of results to expect and the operations that can be performed with the software.

For porting, it is very important to have a good understanding of the UNIX operating system. For example, one should learn the layout of the file system and where to find information. Also, an advanced, thorough understanding of the development process is vital, namely, knowing how the kernel and libraries interact with the linker and compiler.

After the configuration testing, a broad, quick perusal of the source is good for getting a general feel of its layout. At this point, be aware of programming constructs that can cause problems on the target system and make note of them. At this stage, if familiarity with the target system does not exist, then experiments should be conducted on that system to test its capabilities and to get a feel for what it can do.

One method to gain a better understanding of the source code is to add meaningful comments to code sections where they are lacking or absent. Another good way to understand the source code is to change a small section of it and to see how this affects the program. The portmapper by default runs on port 113, a port number in the privileged range that requires additional permissions. To minimize potential quirks necessary to gain privilege and also to help gain familiarity with the portmapper code, the porter can change the portmapper's default port to a number higher than 1024 (the unprivileged port range).

Next, the source code can be run through *gdb* to get a feel for how the program runs. If *gdb* is not available, then *strace* must do. If errors are encountered on the target machine, try to emulate them on the development machine, and stop them at the point of failure.

Another major concern in the porting process was that a lot of security checks are made that do not have a bearing in the target environment. The security can also inhibit understanding the functional part of the ported code. So these checks should be disabled to get the functional part of the port working, then they should be reapplied for the port to become complete. Since a good understanding of the port has now been obtained, it will be easier to understand which of the application level security checks are applicable for the target system.

When the port is divided into separate distinct components that rely on each other, a case could arise when one component is completely ported and another is attempted, and that could reveal problems in the first component that need to be addressed. So there is a cycle of returning to the first component, making changes to it, then testing it, then testing it with the second component to see if the first component has met it needs. For example, to test portmapper, a test harness with essentially all of the functionality of *mountd* and *nfsd* is needed. Therefore instead of creating new test software, these modules were used to test

portmapper. However to know if these modules work correctly, portmapper must work correctly. Thus, all components need to be tested together.

The bulk of the porting effort deals with unimplemented function calls. To more speedily handle this task, a program should be made that scans the servers for all functions and system calls that are used. This list of required functions can then be compared to what is supported on the target system to see what needs to be changed or created.

C. ADDITIONAL WORK

While a working implementation of NFS was successfully made, it does not provide the capability for a user to change his security access level. Instead, it uses the simpler authentication method provided by the NFS protocol, which only has host level granularity. NFS only runs at a single level with the privileges of the user executing it.

A way of approaching an implementation that includes MLS functionality is to add another lower level layer component that will be responsible for authentication. This component must be able to handle security labels in a manner consistent with the XTS-400. Also, the format of the configuration file, "/etc/exports", will need to be extended to support security labels.

This port was accomplished by removing from the ported code all system calls that were not available. A cleaner way of dealing with this is to implement these missing system calls. To do this, instructions for compiling the operating system component are needed, which entails a thorough understanding of the relevant STOP code.

Another future task is to implement NFS inside the Linux kernel layer of the XTS-400. A good starting point for a kernel implementation is to look into the Linux kernel code for the kernel entry point for NFS, which is *sys_nfsservctl* (system call number 169), for an impression of how NFS calls are dealt with. Then a design can be developed based on an analysis of the Linux kernel NFS server, with the requirement that it adheres to MLS constraints of the underlining

XTS-400 kernel. An advantage of a kernel implementation is better performance because there will be fewer context switches from system mode into user mode.

NFS was not built with security in mind. Thus, all of the NFS shared file's data is sent in clear text. NFS should be run in a safe environment where there is physical security to the network cabling wires. To amend the problem of *sniffable* network traffic, a design to couple NFS with encryption is necessary.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

It has been shown that porting is an arduous problem, but steps can be taken to make it tractable. One of the most difficult parts of porting is the learning curve necessary to understand the system and the source code of the ported software. Hindsight proves to be a valuable tool to show which methods were more effective and how this port could have been better approached. In conclusion, the effort necessary to get an important network service to work on an MLS system was manageable, and there should be further attempts to get other popular software onto the XTS-400.

This project has provided a modified library that has implemented some of the missing RPC-related functions for the XTS-400. Additionally, the portmapper can also be used with any other RPC-based server. So, components have been developed that can be reused in the future to quicken the porting process for additional software to the XTS-400.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: SOURCE CODE

A. INTRODUCTION

This section will introduce some of the important source code changes that were made to make the port work. It contains the diff file along with explanation for why it was changed in the comment sections of the code. The '-' lines means that it has been added. The '!' lines means the section of code that is different between the original and new source file.

B. GLIBC DIFF FILES

```
*** pmap_clnt.c   Sun Mar 28 21:20:44 2004
--- /usr/src/redhat/SOURCES/glib/glibc-2.2.93/sunrpc/pmap_clnt.c  Thu
Aug  8 08:28:03 2002
***************
*** 41,114 ****
  #include <sys/ioctl.h>
  #include <sys/socket.h>
  #include <netinet/in.h>
  #include <arpa/inet.h>
  #include <rpc/rpc.h>
- //#include "bar/pmap_prot.h"
  #include <rpc/pmap_prot.h>
  #include <rpc/pmap_clnt.h>
- #include <string.h>

  /*
   * Same as get_myaddress, but we try to use the loopback
   * interface. portmap caches interfaces, and on DHCP clients,
   * it could be that only loopback is started at this time.
   */
-
- /*
-
- This will be the stub function will replace
- __get_myaddress.
-
- */
- static bool_t
- __kandy_get_myaddress ( struct sockaddr_in *addr)
- {
-   int s;
-   char buf[BUFSIZ];
-   struct ifconf ifc;
-   struct ifreq ifreq, *ifr;
-   int len, loopback = 1;
-
-   printf("kdebug: in kandy_get_myaddress\n");
-
-   /*
```

```
-
-   print out the port number that will be used to connect to the
-   portmapper.
-
-   */
-   printf("kdebug glibc: pmap portnumber is: %i\n", PMAPPORT);
-
-
-
- addr->sin_family = 2;
-
- /*
-
-   Set the port number of the portmapper we will be connecting to,
-   need to be in network form
-   so for port 111, use the following: 28416
-
-   for port 1111, use the following: 22276
-
- */
- // addr->sin_port = 28416;
-
-    addr->sin_port = 28416;
-
-   /*
-
-    Set the IP address to 127.0.0.1
-
-   */
-    addr->sin_addr.s_addr = 16777343;
-    memcpy(addr->sin_zero,"\0\0\0\0\0\0\0", 7);
-
-     return TRUE;
- }
-
  static bool_t
  __get_myaddress (struct sockaddr_in *addr)
  {
    int s;
    char buf[BUFSIZ];
--- 41,58 ----
***************
*** 173,195 ****
    int socket = -1;
    CLIENT *client;
    struct pmap parms;
    bool_t rslt;

!   /*
!
!   Call the stub function instead
!
!   */
!   if (!__kandy_get_myaddress (&myaddress))
      return FALSE;
-
- //  comment out the improperly implemented function
```

```
-
- //  if (!__get_myaddress (&myaddress))
- //    return FALSE;
-
    client = INTUSE(clntudp_bufcreate) (&myaddress, PMAPPROG, PMAPVERS,
                                        timeout, &socket, RPCSMALLMSGSIZE,
                                        RPCSMALLMSGSIZE);
    if (client == (CLIENT *) NULL)
      return (FALSE);
--- 117,128 ----
    int socket = -1;
    CLIENT *client;
    struct pmap parms;
    bool_t rslt;

!   if (!__get_myaddress (&myaddress))
      return FALSE;
    client = INTUSE(clntudp_bufcreate) (&myaddress, PMAPPROG, PMAPVERS,
                                        timeout, &socket, RPCSMALLMSGSIZE,
                                        RPCSMALLMSGSIZE);
    if (client == (CLIENT *) NULL)
      return (FALSE);
***************
*** 221,243 ****
    int socket = -1;
    CLIENT *client;
    struct pmap parms;
    bool_t rslt;

!   /*
!
!   This function is called in pmap_unset,
!   which is one of the function called to communicate
!   with the portmapper.
!
!   */
!   if (!__kandy_get_myaddress (&myaddress))
!     return FALSE;
!
! //  if (!__get_myaddress (&myaddress))
! //    return FALSE;
!
    client = INTUSE(clntudp_bufcreate) (&myaddress, PMAPPROG, PMAPVERS,
                                        timeout, &socket, RPCSMALLMSGSIZE,
                                        RPCSMALLMSGSIZE);
    if (client == (CLIENT *) NULL)
      return FALSE;
--- 154,165 ----
    int socket = -1;
    CLIENT *client;
    struct pmap parms;
    bool_t rslt;

!   if (!__get_myaddress (&myaddress))
!     return FALSE;
    client = INTUSE(clntudp_bufcreate) (&myaddress, PMAPPROG, PMAPVERS,
                                        timeout, &socket, RPCSMALLMSGSIZE,
```

```
                                RPCSMALLMSGSIZE);
    if (client == (CLIENT *) NULL)
      return FALSE;


*** clnt_udp.c    2004-03-22 07:27:23.000000000 -0800
--- /usr/src/redhat/SOURCES/glib/glibc-2.2.93/sunrpc/clnt_udp.c   2002-
08-08 01:39:50.000000000 -0700
***************
*** 191,230 ****
        goto fooy;
      }
        /* attempt to bind to prov port */
        (void) bindresvport (*sockp, (struct sockaddr_in *) 0);
        /* the sockets rpc controls are non-blocking */
-
-     /*
-
-     This is to test to see if FIOBIO is implemented
-     with ioctl.
-
-     */
-     printf("before ioctl FIOBIO\n");
-
        (void) __ioctl (*sockp, FIONBIO, (char *) &dontblock);
-
-     printf("after ioctl FIOBIO\n");
-
-
  #ifdef IP_RECVERR
        {
      int on = 1;
-
-     /*
-
-     This is to test to see if SOL_IP is implemented
-     for setsockopt.
-
-     */
-     printf("before setsockopt SOL_IP, IP_RECVERR\n");
-
      __setsockopt (*sockp, SOL_IP, IP_RECVERR, &on, sizeof(on));
-
-     printf("after setsockopt SOL_IP, IP_RECVERR\n");
-
        }
  #endif
        cu->cu_closeit = TRUE;
      }
    else
--- 191,205 ----


*** svc.c   2004-03-22 07:59:40.000000000 -0800
--- /usr/src/redhat/SOURCES/glib/glibc-2.2.93/sunrpc/svc.c  2002-08-08
01:39:50.000000000 -0700
***************
```

```
*** 379,396 ****
  svc_getreq_poll (struct pollfd *pfdp, int pollretval)
  {
    register int i;
    register int fds_found;

-   /*
-
-   svc_getreq_poll is used to the querty the RPC subsystem
-   to see if any messages have arrived.
-
-   */
-   printf("kdebug: in svc_getreq_poll\n");
-
    for (i = fds_found = 0; i < svc_max_pollfd && fds_found <
pollretval; ++i)
      {
        register struct pollfd *p = &pfdp[i];

        if (p->fd != -1 && p->revents)
--- 379,388 ----
***************
*** 402,418 ****
          xprt_unregister (xports[p->fd]);
        else
          INTUSE(svc_getreq_common) (p->fd);
      }
      }
-
-    /*
-
-    svc_getreq_poll is finished processing the message.
-
-    */
-    printf("kdebug: done svc_getreq_poll\n");
  }
  INTDEF (svc_getreq_poll)


  void
--- 394,403 ----
***************
*** 423,439 ****
    register SVCXPRT *xprt;
    char cred_area[2 * MAX_AUTH_BYTES + RQCRED_SIZE];
    msg.rm_call.cb_cred.oa_base = cred_area;
    msg.rm_call.cb_verf.oa_base = &(cred_area[MAX_AUTH_BYTES]);

-   /*
-
-   Authenication is done in this function.
-
-   */
-   printf("kdebug: in svc_getreq_common\n"); // kandy
-
    xprt = xports[fd];
    /* Do we control fd? */
```

```
    if (xprt == NULL)
       return;

--- 408,417 ----
***************
*** 501,518 ****
        else
          svcerr_noprog (xprt);
        /* Fall through to ... */
      }
      call_done:
-
-     /*
-
-     Now the message is done through the authenication phase.
-
-     */
-     printf("kdebug: at call_done\n"); // kandy
-
        if ((stat = SVC_STAT (xprt)) == XPRT_DIED)
      {
        SVC_DESTROY (xprt);
        break;
      }
--- 479,488 ----
```

C. PORTMAPPER DIFF FILES

```
*** portmap.c     2004-03-22 08:35:58.000000000 -0800
--- /home/tux/portmap/stock/portmap_5beta/portmap.c   1996-07-06
14:06:24.000000000 -0700
***************
*** 88,106 ****
  #include <sys/socket.h>
  #include <sys/ioctl.h>
  #include <sys/wait.h>
  #include <sys/signal.h>
  #include <sys/time.h>
-
-
- #include <fcntl.h>
- /*
- #include <unistd.h>
- #include <sys/types.h>
- #include <sys/stat.h>
- */
-
  #include <sys/resource.h>
  #ifdef SYSV40
  #include <netinet/in.h>
  #endif

--- 88,97 ----
***************
```

```
*** 155,241 ****
  #ifndef INADDR_LOOPBACK
  #define INADDR_LOOPBACK ntohl(inet_addr("127.0.0.1"))
  #endif
  #endif

-
- #define syslog(a,b) kandy_syslog(a, b)
- #include "kandydbg.h"
-
- int logfilefd = 0;
- int kandbgfd;
-
- int kandy_syslog(type,bufp )
- int type; char *bufp; {
-
-     int len = 0;
-
- /*
-     len = strlen(bufp);
-
-     write(logfilefd, bufp, len);
-     write(logfilefd, "\n", 1);
- */
- //  write(logfilefd, "there was an error\n", 19);
- //  write(logfilefd, "it shouldn't die here\n", 19);
-     dprintf(logfilefd, "in kandy's syslog print\n");
-
-     return len;
- }
-
-
-
  main(argc, argv)
      int argc;
      char **argv;
  {
      SVCXPRT *xprt;
      int sock, c;
      struct sockaddr_in addr;
      int len = sizeof(struct sockaddr_in);
      register struct pmaplist *pml;
- int kandret = 0;
-
-
- logfilefd =     open("/tmp/portmaplog", O_RDWR|O_CREAT|O_APPEND,
S_IRWXU);
-
-
- if (logfilefd < 0 ) {
- printf("can't open log file portmaplog\n");
- exit(1);
- }
-
- //write(logfilefd, "starting log file portmaplog\n", 18);
-
- // dprintf(logfilefd, "starting log file portmaplog\n");
```

```
-
- // syslog(2,"foo");
-
- /*
-
- Redirect stdin, stdout, stderr to a log file,
- so that the server can be run as a daemon.
-
- */
- close(0);
- close(1);
- close(2);
- dup2(logfilefd, 0);
- dup2(logfilefd, 1);
- dup2(logfilefd, 2);
-
- /*
- kandbgfd =      open("/tmp/kandydbg", O_RDWR|O_CREAT|O_APPEND,
S_IRWXU);
-
- if (kandbgfd < 0 ) {
- printf("can't open log file kandydbg\n");
- exit(1);
- }
- */
-
- //        dprintf(kandbgfd, "staring log file\n");
-

      while ((c = getopt(argc, argv, "dv")) != EOF) {
            switch (c) {

            case 'd':
--- 146,164 ----
***************
*** 252,279 ****
                  (void) fprintf(stderr, "-v: verbose logging\n");
                  exit(1);
            }
      }

- write(logfilefd, "before daemon\n", 14);
-           dprintf(kandbgfd, "before daemon\n");
-
      if (!debugging && daemon(0, 0)) {
! //        (void) fprintf(stderr, "portmap: fork: %s",
strerror(errno));
!           dprintf(kandbgfd, "portmap: fork: %s", strerror(errno));
            exit(1);
      }
-           dprintf(kandbgfd, "after daemon\n");

! #ifdef FACILITY
!     openlog("portmap", debugging ? LOG_PID | LOG_PERROR : LOG_PID,
FACILITY);
  #else
      openlog("portmap", debugging ? LOG_PID | LOG_PERROR : LOG_PID);
```

```
  #endif
- write(logfilefd, "before socket\n", 14);
-
-

      if ((sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
            syslog(LOG_ERR, "cannot create udp socket: %m");
            exit(1);
      }
--- 175,195 ----
                  (void) fprintf(stderr, "-v: verbose logging\n");
                  exit(1);
            }
      }

      if (!debugging && daemon(0, 0)) {
!           (void) fprintf(stderr, "portmap: fork: %s",
strerror(errno));
            exit(1);
      }

! #ifdef LOG_MAIL
!     openlog("portmap", debugging ? LOG_PID | LOG_PERROR : LOG_PID,
!         FACILITY);
  #else
      openlog("portmap", debugging ? LOG_PID | LOG_PERROR : LOG_PID);
  #endif

      if ((sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
            syslog(LOG_ERR, "cannot create udp socket: %m");
            exit(1);
      }
***************
*** 287,297 ****
      addr.sin_port = htons(PMAPPORT);
      if (bind(sock, (struct sockaddr *)&addr, len) != 0) {
            syslog(LOG_ERR, "cannot bind udp: %m");
            exit(1);
      }
- write(logfilefd, "before socket\n", 14);

      if ((xprt = svcudp_create(sock)) == (SVCXPRT *)NULL) {
            syslog(LOG_ERR, "couldn't do udp_create");
            exit(1);
      }
--- 203,212 ----
***************
*** 381,392 ****
  void
  perror(what)
      const char *what;
  {

!     //syslog(LOG_ERR, "%s: %m", what);
!     syslog(LOG_ERR, "%s: %m");
  }
  #endif
```

```
  static struct pmaplist *
  find_service(prog, vers, prot)
--- 296,306 ----
  void
  perror(what)
      const char *what;
  {

!     syslog(LOG_ERR, "%s: %m", what);
  }
  #endif

  static struct pmaplist *
  find_service(prog, vers, prot)
***************
*** 743,759 ****
      /*
       * fork a child to do the work.  Parent immediately returns.
       * Child exits upon completion.
       */
      if ((pid = fork()) != 0) {
!           if (pid < 0) {
!
!                 //syslog(LOG_ERR, "CALLIT (prog %lu): fork: %m",
!                  //   a.rmt_prog);
!
!                 syslog(LOG_ERR, "CALLIT (prog %lu): fork: %m");
! }
            return;
      }
      port = pml->pml_map.pm_port;
      get_myaddress(&me);
      me.sin_port = htons(port);
--- 657,669 ----
      /*
       * fork a child to do the work.  Parent immediately returns.
       * Child exits upon completion.
       */
      if ((pid = fork()) != 0) {
!           if (pid < 0)
!                 syslog(LOG_ERR, "CALLIT (prog %lu): fork: %m",
!                     a.rmt_prog);
            return;
      }
      port = pml->pml_map.pm_port;
      get_myaddress(&me);
      me.sin_port = htons(port);


*** pmap_check.c  Sun Mar 28 21:26:09 2004
--- /home/tux/portmap/stock/portmap_5beta/pmap_check.c      Sun Jul  7
08:49:10 1996
***************
*** 35,46 ****
  #ifndef lint
  static char sccsid[] = "@(#) pmap_check.c 1.8 96/07/07 10:49:10";
```

```
  #endif

  #include <rpc/rpc.h>
! //#include <rpc/pmap_prot.h>
! #include "pmap_prot.h"
  #include <syslog.h>
  #include <netdb.h>
  #include <sys/signal.h>
  #ifdef SYSV40
  #include <netinet/in.h>
--- 35,45 ----
  #ifndef lint
  static char sccsid[] = "@(#) pmap_check.c 1.8 96/07/07 10:49:10";
  #endif

  #include <rpc/rpc.h>
! #include <rpc/pmap_prot.h>
  #include <syslog.h>
  #include <netdb.h>
  #include <sys/signal.h>
  #ifdef SYSV40
  #include <netinet/in.h>
***************
*** 100,124 ****

      /*
       * Give up root privileges so that we can never allocate a
privileged
       * port when forwarding an rpc request.
       */
-
-     /*
-
-     setuid is not implemented properly on the XTS-400
-     so it is not used.
-
-     */
-
- /*
      if (setuid(1) == -1) {
      syslog(LOG_ERR, "setuid(1) failed: %m");
      exit(1);
      }
- */
-
      (void) signal(SIGINT, toggle_verboselog);
  }

  /* check_default - additional checks for NULL, DUMP, GETPORT and
unknown */

--- 99,112 ----
***************
*** 167,197 ****
  u_long  prog;
  u_long  port;
  {
```

```
      struct sockaddr_in *addr = svc_getcaller(xprt);

- /*
-
- The privileged port check is not applicable
- on the XTS-400, so it is not used.
-
- */
- #ifdef KANDY_SEC_TEST
-
      if (xprt != ludp_xprt && xprt != ltcp_xprt) {
  #ifdef HOSTS_ACCESS
      (void) good_client(addr);           /* because of side effects */
  #endif
      log_bad_owner(addr, proc, prog);
      return (FALSE);
      }
      if (port && !check_privileged_port(addr, proc, prog, port))
      return (FALSE);
-
- #endif
-     printf("kandy, in check_setuset\n");
-
      if (verboselog)
      log_client(addr, proc, prog);
      return (TRUE);
  }

--- 155,173 ----
***************
*** 201,232 ****
  struct sockaddr_in *addr;
  u_long  proc;
  u_long  prog;
  u_long  port;
  {
-
- /*
-
- The privileged port check is not applicable
- on the XTS-400, so it is not used.
-
- */
- #ifdef KANDY_SEC_TEST
-
      if (!from_local(addr)) {
  #ifdef HOSTS_ACCESS
      (void) good_client(addr);           /* because of side effects */
  #endif
      log_bad_owner(addr, proc, prog);
      return (FALSE);
      }
      if (port && !check_privileged_port(addr, proc, prog, port))
      return (FALSE);
-
- #endif
-
```

```
-     printf("kandy2, in check_setuset\n");
      if (verboselog)
      log_client(addr, proc, prog);
      return (TRUE);
  }

--- 177,195 ----

*** from_local.c  Sun Mar 28 21:26:43 2004
--- /home/tux/portmap/stock/portmap_5beta/from_local.c      Fri May 31
13:52:58 1996
***************
*** 104,163 ****
      struct ifreq *the_end;
      int     sock;
      char    buf[BUFSIZ];

      /*
-
-     This struct is used to return the local IP address
-
-     */
-     struct in_addr localinaddr;
-
-     /*
-
-     Set the s_addr structure to "127.0.0.1"
-
-     */
-     localinaddr.s_addr = 16777343;
-
-
-     /*
-
-     If the global addrs array which contains the information
-     of what the server should set it IP address is not yet
-     set, then set it.
-
-     */
-     if (num_addrs < 1) {
-           grow_addrs();
-           //addrs[0].sin_family =  2;
-           addrs[0] = localinaddr;
-           //memcpy(addrs[0].sin_zero , "\0\0\0\0\0\0\0", 7);
-           num_addrs = 1;
-     }
-
-     return 1;
-
- }
-
- /*
-
- This is the original find_local function which does not function
- properly.  It is commented out so that the replaced version is used.
-
- */
```

```
- #ifdef KANDY_SEC_TEST
- find_local()
- {
-     struct ifconf ifc;
-     struct ifreq ifreq;
-     struct ifreq *ifr;
-     struct ifreq *the_end;
-     int     sock;
-     char    buf[BUFSIZ];
-
-     /*
       * Get list of network interfaces. We use a huge buffer to allow
for the
       * presence of non-IP interfaces.
       */

      if ((sock = socket(PF_INET, SOCK_DGRAM, 0)) < 0) {
--- 104,113 ----
***************
*** 200,211 ****
      }
      (void) close(sock);
      return (num_local);
  }

- #endif
-
  /* from_local - determine whether request comes from the local system
*/

  from_local(addr)
  struct sockaddr_in *addr;
  {
--- 150,159 ----


*** daemon.c      2004-03-22 08:14:51.000000000 -0800
--- /home/tux/portmap/stock/portmap_5beta/daemon.c    1992-06-11
13:53:12.000000000 -0700
***************
*** 43,91 ****
  #define STDERR_FILENO 2

  /* From paths.h */
  #define _PATH_DEVNULL "/dev/null"

- /*
-
- This header file contains necessary
- function for output message that are helpful
- for debugging.
-
- */
- #include "kandydgb.h"
-
  daemon(nochdir, noclose)
      int nochdir, noclose;
```

```
  {
      int cpid;

-     /*
-
-     This function is enter when the portmapper is run without
-     the foreground flag.
-
-     */
-     dprintf(kandbgfd, "in daemon() \n");
-
      if ((cpid = fork()) == -1)
            return (-1);
      if (cpid)
            exit(0);
-
-     /*
-
-     This is use to see if the setsid will work
-     on the XTS system.
-
-     */
-     dprintf(kandbgfd, "before setsid() \n");
-
      (void) setsid();
-
-     dprintf(kandbgfd, "after setsid() \n");
-
      if (!nochdir)
            (void) chdir("/");
      if (!noclose) {
            int devnull = open(_PATH_DEVNULL, O_RDWR, 0);

--- 43,62 ----
```

D. MOUNTD AND NFSD DIFF FILES

```
*** daemon.c      2004-03-22 08:14:51.000000000 -0800
--- /home/tux/portmap/stock/portmap_5beta/daemon.c    1992-06-11
13:53:12.000000000 -0700
***************
*** 43,91 ****
  #define STDERR_FILENO 2

  /* From paths.h */
  #define _PATH_DEVNULL "/dev/null"

- /*
-
- This header file contains necessary
- function for output message that are helpful
- for debugging.
-
```

```
- */
- #include "kandydgb.h"
-
  daemon(nochdir, noclose)
      int nochdir, noclose;
  {
      int cpid;

-     /*
-
-     This function is enter when the portmapper is run without
-     the foreground flag.
-
-     */
-     dprintf(kandbgfd, "in daemon() \n");
-
      if ((cpid = fork()) == -1)
            return (-1);
      if (cpid)
            exit(0);
-
-     /*
-
-     This is use to see if the setsid will work
-     on the XTS system.
-
-     */
-     dprintf(kandbgfd, "before setsid() \n");
-
      (void) setsid();
-
-     dprintf(kandbgfd, "after setsid() \n");
-
      if (!nochdir)
            (void) chdir("/");
      if (!noclose) {
            int devnull = open(_PATH_DEVNULL, O_RDWR, 0);

--- 43,62 ----


*** fh.c    Sat Mar 27 22:44:07 2004
--- /home/tux/nfs/stock/nfs-server-2.2beta46/fh.c     Mon Nov 23
13:08:11 1998
***************
*** 143,154 ****
  static void     fh_unlink_fdcache(fhcache *fhc);

  static void
  fh_move_to_front(fhcache *fhc)
  {
- printf("kdebug: in fh_move_to_front\n");
-
      /* Remove from current posn */
      fhc->prev->next = fhc->next;
      fhc->next->prev = fhc->prev;
```

```
      /* Insert at head */
--- 143,152 ----
***************
*** 178,188 ****

  static fhcache *
  fh_lookup(psi_t psi)
  {
      register fhcache *fhc;
- printf("kdebug: in fh_lookup\n");

      fhc = fh_hashed[psi % HASH_TAB_SIZE];
      while (fhc != NULL && fhc->h.psi != psi)
            fhc = fhc->hash_next;
      return (fhc);
--- 176,185 ----
***************
*** 536,560 ****
      char        *slash_stack[HP_LEN];
      struct stat sbuf;
      psi_t       psi;
      int         i;

- int y = 0;
- printf("kdebug: in fh_buildpath, dump: %s\n", fh_dump(h));
-
      if (h->hash_path[0] >= HP_LEN) {
            Dprintf(L_ERROR, "impossible hash_path[0] value: %s\n",
                              fh_dump(h));
            return NULL;
      }

- printf("kdebug: before efs_stat 1\n");
      if (efs_stat("/", &sbuf) < 0)
            return (NULL);
      psi = pseudo_inode(sbuf.st_ino, sbuf.st_dev);
      if (h->hash_path[0] == 0) {
- printf("kdebug: hash_path[0] is zero\n");
            if (psi != h->psi)
                  return (NULL);
            return xstrdup("/");
      }
      /* else */
--- 533,552 ----
***************
*** 562,675 ****
            return (NULL);

      auth_override_uid(ROOT_UID);  /* for x-only dirs */
      strcpy(pathbuf, "/");
      cookie_stack[2] = 0;
- y = 0;
      for (i = 2; i <= h->hash_path[0] + 1; i++) {
            DIR *dir;
            struct dirent *dp;

      backtrack:
```

```
- y++;
-
- printf("kdebug: before efs_stat 2, cookie is:%i\n", cookie_stack[i]);
- printf("kdebug: pathbuf is: %s\n", pathbuf);
-
- /*
-
- This is necessary to detect an infinite loop because a different
semantic
- implementation of seekdir and readdir. It will return with an error
- is an infinite loop is detected.
-
- */
- if (y > 5) {
- printf("kdebug: looks like going to be stuck in for loop,
leaving\n");
-
- return (NULL);
- }
-
            if (efs_stat(pathbuf, &sbuf) >= 0
                && (dir = efs_opendir(pathbuf)) != NULL) {
!                 if (cookie_stack[i] != 0){
!                 printf("kdebug: cookie setting, index:%i\n",
cookie_stack[i]);
                      efs_seekdir(dir, cookie_stack[i]);
- /*
-
- The way the combination of seekdir works with readdir, is that
- readdir will start reading from the entry set by seekdir.
- This is different from the way Linux work, which has readdir
- start after the offset set by seekdir. That's why an extra
- readdir is needed here. This can be tested when there is hash
- collision when trying to resolve a directory name.
-
- */
-                     efs_readdir(dir); // eat an entry
- }
                  while ((dp = efs_readdir(dir))) {
- printf("kdebug: ck:%i, dp->dname is: %s\n", dp->d_off, dp->d_name);
                      if (strcmp(dp->d_name, ".") != 0
!                         && strcmp(dp->d_name, "..") != 0
!
! //&& strcmp(dp->d_name, "dev") != 0
!
! ) {
! // add for debugging, remove later
                            psi = pseudo_inode(dp->d_ino,
sbuf.st_dev);
- printf("i: %i, h0-%i, h0x-%x,,hpi:%x, ", i,  h->hash_path[0], h-
>hash_path[0], h->hash_path[i] );
- printf("^^psi %x, h->psi %x, hash %x,,", psi, h->psi, hash_psi(psi));
- printf("i: %i, :d_ino-%i st_dev-%i|\n",
- i, dp->d_ino, sbuf.st_dev );
                            if (i == h->hash_path[0] + 1) {
                                  if (psi == h->psi) {
                                        /*GOT IT*/
```

```
                                        strcat(pathbuf, dp->d_name);
- printf("kdebug: pathbuf 2 is: %s\n", pathbuf);
                                        path = xstrdup(pathbuf);
                                        efs_closedir(dir);
                                        auth_override_uid(auth_uid);
                                        return (path);
                                    }
                                } else {
- printf("kdebug: in else of pathbuf 2, is: %s\n", pathbuf);
-
-
                                    if (hash_psi(psi) == h-
>hash_path[i]) {
                                        /*PERHAPS WE'VE GOT IT */
                                        cookie_stack[i] =
efs_telldir(dir);
                                        cookie_stack[i + 1] = 0;
                                        slash_stack[i] = pathbuf +
strlen(pathbuf);
                                        strcpy(slash_stack[i], dp-
>d_name);
                                        strcat(pathbuf, "/");
- printf("kdebug: ck:%i, pathbuf 3 is now: %s, %x, %x\n",
cookie_stack[i], pathbuf, h->hash_path[i], hash_psi(psi));

                                        efs_closedir(dir);
                                        goto deeper;
                                    }
                                }
                            }
                        }
                        /* dp == NULL */
                        efs_closedir(dir);
-           } else {
- if (efs_stat(pathbuf, &sbuf) < 0) printf("kdebug: efs_stat is <0 in
else\n");
- if (dir == NULL) printf("kdebug: dir is null in else\n");
-
-
            }
-
            /* shallower */
            i--;
            if (i < 2) {
- printf("kdebug: search exhausted\n");
                auth_override_uid(auth_uid);
                return (NULL);    /* SEARCH EXHAUSTED */
            }

            /* Prune path */
            *(slash_stack[i]) = '\0';
- printf("kdebug: it was pruned, path is:%s\n", pathbuf);
            goto backtrack;
      deeper:
            ;
      }
      auth_override_uid(auth_uid);
```

```
--- 554,612 ----
            return (NULL);

      auth_override_uid(ROOT_UID);  /* for x-only dirs */
      strcpy(pathbuf, "/");
      cookie_stack[2] = 0;
      for (i = 2; i <= h->hash_path[0] + 1; i++) {
            DIR *dir;
            struct dirent *dp;

      backtrack:
            if (efs_stat(pathbuf, &sbuf) >= 0
                && (dir = efs_opendir(pathbuf)) != NULL) {
!                 if (cookie_stack[i] != 0)
                        efs_seekdir(dir, cookie_stack[i]);
                  while ((dp = efs_readdir(dir))) {
                        if (strcmp(dp->d_name, ".") != 0
!                           && strcmp(dp->d_name, "..") != 0) {
                              psi = pseudo_inode(dp->d_ino,
sbuf.st_dev);
                              if (i == h->hash_path[0] + 1) {
                                    if (psi == h->psi) {
                                          /*GOT IT*/
                                          strcat(pathbuf, dp->d_name);
                                          path = xstrdup(pathbuf);
                                          efs_closedir(dir);
                                          auth_override_uid(auth_uid);
                                          return (path);
                                    }
                              } else {
                                    if (hash_psi(psi) == h-
>hash_path[i]) {
                                          /*PERHAPS WE'VE GOT IT */
                                          cookie_stack[i] =
efs_telldir(dir);
                                          cookie_stack[i + 1] = 0;
                                          slash_stack[i] = pathbuf +
strlen(pathbuf);
                                          strcpy(slash_stack[i], dp-
>d_name);
                                          strcat(pathbuf, "/");

                                          efs_closedir(dir);
                                          goto deeper;
                                    }
                              }
                        }
                  }
                  /* dp == NULL */
                  efs_closedir(dir);
            }
            /* shallower */
            i--;
            if (i < 2) {
                  auth_override_uid(auth_uid);
                  return (NULL);    /* SEARCH EXHAUSTED */
            }
```

```
            /* Prune path */
            *(slash_stack[i]) = '\0';
            goto backtrack;
      deeper:
            ;
      }
      auth_override_uid(auth_uid);
***************
*** 733,743 ****
            Dprintf(L_ERROR, "impossible hash_path[0] value: %s\n",
                              fh_dump(h));
            return NULL;
      }

- printf("kdebug: before efs_stat 3\n");
      if (efs_stat("/", &sbuf) < 0)
            return (NULL);
      psi = pseudo_inode(sbuf.st_ino, sbuf.st_dev);

      if (h->hash_path[0] == 0) {
--- 670,679 ----
***************
*** 753,763 ****
      i = 2;
      cookie_stack[i] = 0;
      while (i <= h->hash_path[0] + 1) {
            DIR *dir;

- printf("kdebug: before efs_stat 4\n");
            if (efs_stat(pathbuf, &sbuf) >= 0
                && (dir = efs_opendir(pathbuf)) != NULL) {
                  if (cookie_stack[i] != 0)
                        efs_seekdir(dir, cookie_stack[i]);
                  if (!fh_buildcomp(h, sbuf.st_dev, dir, i, pathbuf)) {
--- 689,698 ----
***************
*** 877,888 ****
  fh_find(svc_fh *h, int mode)
  {
      register fhcache *fhc, *flush;
      int          check;

- printf("kdebug: in fh_find\n");
-
      check = (mode & FHFIND_CHECK);
      mode &= 0xF;

  #ifdef FHTRACE
      if (h->hash_path[0] >= HP_LEN) {
--- 812,821 ----
***************
*** 914,924 ****
             */
            if (check) {
                  struct stat *s = &fhc->attrs;
                  psi_t       psi;
```

```
                  nfsstat           dummy;
- printf("kdebug: fh_find, rebuilding path for file\n");

                  if (efs_lstat(fhc->path, s) < 0) {
                        Dprintf(D_FHTRACE,
                              "fh_find: stale fh: lstat: %m\n");
                  } else {
--- 847,856 ----


*** logging.c     2004-03-22 09:02:45.000000000 -0800
--- /home/tux/nfs/stock/nfs-server-2.2beta46/logging.c      1998-10-30
08:11:22.000000000 -0800
***************
*** 13,25 ****
   *        as is, with no warranty expressed or implied.
   */

  #include "nfsd.h"

- // This is necessary to compile successfully
- #include <time.h>
-
  #ifdef HAVE_SYSLOG_H
  #include <syslog.h>
  #else
  #define LOG_FILE      "/var/tmp/%s.log"
  #endif
--- 13,22 ----


*** mountd.c      2004-03-20 08:11:52.000000000 -0800
--- /home/tux/nfs/stock/nfs-server-2.2beta46/mountd.c 1999-06-02
05:10:33.000000000 -0700
***************
*** 75,87 ****
  static char     *program_name;
  int       need_reinit = 0;
  int       need_flush = 0;
  extern char     version[];

- // file descriptor for log file
- int kandbgfd;
-
  /*
   * NULL
   * Do nothing
   */
  void *
--- 75,84 ----
***************
*** 132,172 ****

      /* Now authenticate the intruder... */
      if (((cp = auth_clnt(rqstp)) == NULL)
        || (mp = auth_path(cp, rqstp, argbuf)) == NULL
        || mp->o.noaccess) {
```

```
-
-
- printf("kdebug: in authenticaton\n") ; // kandy
-
            res->fhs_status = NFSERR_ACCES;
  #ifdef WANT_LOG_MOUNTS
            Dprintf(L_WARNING, "Blocked attempt of %s to mount %s\n",
                                inet_ntoa(addr), argbuf);
  #endif /* WANT_LOG_MOUNTS */
            Dprintf (D_CALL, "\tmount res = %d\n", res->fhs_status);
            return (res);
       }
- printf("kdebug: after blocking authenticaton\n") ; // kandy

       /* Check the file. We can now return valid results to the
        * client. */
       if ((errno = saved_errno) != 0 || stat(argbuf, &stbuf) < 0) {
            res->fhs_status = nfs_errno();
            Dprintf (D_CALL, "\tmount res = %d\n", res->fhs_status);
-
-
- printf("kdebug: returning with errno\n") ; // kandy
-
            return (res);
       }

       if (!S_ISDIR(stbuf.st_mode) && !S_ISREG(stbuf.st_mode)) {
- printf("kdebug: nfserr_notdir\n") ; // kandy
            res->fhs_status = NFSERR_NOTDIR;
       } else if (!re_export && nfsmounted(argbuf, &stbuf)) {
- printf("kdebug: nfserr_acces\n") ; // kandy
            res->fhs_status = NFSERR_ACCES;
       } else {
            res->fhs_status = fh_create((nfs_fh *)
                  &(res->fhstatus_u.fhs_fhandle), argbuf);
            rmtab_add_client(argbuf, rqstp);
--- 129,158 ----
***************
*** 174,184 ****
            Dprintf(L_NOTICE, "%s has been mounted by %s\n",
                               argbuf, inet_ntoa(addr));
  #endif /* WANT_LOG_MOUNTS */
       }
       Dprintf (D_CALL, "\tmount res = %d\n", res->fhs_status);
- printf("kdebug: after mounted\n") ; // kandy
       return (res);
  }

  /*
   * DUMP
--- 160,169 ----
***************
*** 324,349 ****
       int port = 0;
       int c;

       program_name = argv[0];
```

```
-
- kandbgfd =      open("/tmp/mountdlog", O_RDWR|O_CREAT|O_APPEND,
S_IRWXU);
-
- if (kandbgfd < 0 ) {
- printf("can't open log file mountdlog\n");
- exit(1);
- }
-
- close(0);
- close(1);
- close(2);
- dup2(kandbgfd, 0);
- dup2(kandbgfd, 1);
- dup2(kandbgfd, 2);
-
-
      /* Parse the command line options and arguments. */
      opterr = 0;
      while ((c = getopt_long(argc, argv, shortopts, longopts, NULL))
!= EOF)
            switch (c) {
            case 'F':
--- 309,318 ----


*** nfsd.c  2004-03-20 08:11:50.000000000 -0800
--- /home/tux/nfs/stock/nfs-server-2.2beta46/nfsd.c   1999-08-29
08:19:29.000000000 -0700
***************
*** 46,57 ****
  static char pathbuf_1[NFS_MAXPATHLEN + NFS_MAXNAMLEN + 1];

  extern char version[];
  static char *program_name;

- int kpnfsfd;
-
  /*
   * Option table
   */
  static struct option longopts[] = {
        { "auth-deamon",            required_argument,      0,    'a'
},
--- 46,55 ----
***************
*** 119,138 ****
  auth_fh(struct svc_req *rqstp, nfs_fh *fh, nfsstat *statp, int flags)
  {
      static int  total = 0, cached = 0;
      fhcache           *fhc;

- printf("kdebug: in auth_fh\n"); // kandy
-
      /* Try to map FH. If not cached, reconstruct path with root priv
*/
```

```
      fhc = fh_find((svc_fh *)fh, FHFIND_FEXISTS|FHFIND_CHECK);
      if (fhc == NULL) {
            *statp = NFSERR_STALE;
- printf("kdebug: returning nfserr_stale\n"); // kandy
            return NULL;
      }
- printf("kdebug: after fh_find\n"); // kandy

      /* Try to retrieve last client who accessed this fh */
      if (nfsclient == NULL) {
            struct in_addr    caddr;

--- 117,132 ----
***************
*** 140,159 ****
            if (fhc->last_clnt != NULL &&
                fhc->last_clnt->clnt_addr.s_addr == caddr.s_addr) {
                  nfsclient = fhc->last_clnt;
            } else if ((nfsclient = auth_clnt(rqstp)) == NULL) {
                  *statp = NFSERR_ACCES;
- printf("kdebug: returning nfserr_acces 1 in auth_fh\n");
                  return NULL;
            }
      }

      if (fhc->last_clnt == nfsclient) {
            nfsmount = fhc->last_mount; /* get cached mount point */
            cached++;
      } else {
- printf("kdebug: before auth_path\n");
            nfsmount = auth_path(nfsclient, rqstp, fhc->path);
            if (nfsmount == NULL) {
                  *statp = NFSERR_ACCES;
                  return NULL;
            }
--- 134,151 ----
***************
*** 171,191 ****
          ((flags & CHK_NOACCESS) || strcmp(nfsmount->path, fhc-
>path))) {
            struct in_addr    addr = svc_getcaller(rqstp->rq_xprt)-
>sin_addr;
            Dprintf(L_WARNING, "client %s tried to access %s
(noaccess)\n",
                        inet_ntoa(addr), fhc->path);
            *statp = NFSERR_ACCES;
- printf("kdebug: returning nfserr_acces\n");
            return NULL;
      }

      if ((flags & CHK_WRITE) && (nfsmount->o.read_only || read_only))
{
            *statp = NFSERR_ROFS;
- printf("kdebug: returning nfserr_rofs\n"); // kandy
            return NULL;
      }
```

```
- printf("kdebug: before auth_user\n"); // kandy
      auth_user(nfsmount, rqstp);

      *statp = NFS_OK;
      return fhc;
  }
--- 163,180 ----
***************
*** 255,273 ****
  nfsd_nfsproc_getattr_2(nfs_fh *argp, struct svc_req *rqstp)
  {
      nfsstat status;
      fhcache *fhc;

- printf("kdebug: in nfsd_nfsproc_getattr_2\n"); // kandy
- printf("kdebug: argp(nfs_fh) is %s\n", argp->data);
-
      fhc = auth_fh(rqstp, argp, &status, CHK_READ);
      if (fhc == NULL)
            return status;

- printf("kdebug: fhc is not NULL\n"); // kandy
-
      return (fhc_getattr(fhc, &result.attrstat.attrstat_u.attributes,
                                      NULL, rqstp));
  }

  int
--- 244,257 ----
***************
*** 985,1011 ****
      int   c;
  #ifdef MULTIPLE_SERVERS
      int   i, ncopies = 1;
  #endif

- // int kpnfsfd;
-
-
- kpnfsfd =      open("/tmp/nfsdlog", O_RDWR|O_CREAT|O_APPEND,
S_IRWXU);
-
- if (kpnfsfd < 0 ) {
- printf("can't open log file nfsdlog\n");
- exit(1);
- }
-
- close(0);
- close(1);
- close(2);
- dup2(kpnfsfd, 0);
- dup2(kpnfsfd, 1);
- dup2(kpnfsfd, 2);
-
      program_name = argv[0];
      chdir("/");
```

```
      /* Parse the command line options and arguments. */
      opterr = 0;
--- 969,978 ----


*** nfsmounted.c  2004-03-22 09:07:13.000000000 -0800
--- /home/tux/nfs/stock/nfs-server-2.2beta46/nfsmounted.c   1997-12-12
06:03:52.000000000 -0800
***************
*** 59,78 ****
  #endif

  int
  nfsmounted(const char *path, struct stat *sbp)
  {
-
- /*
-
- major is not implemented properly on the XTS-400,
- so it is not used.
-
- */
- #ifdef KANDY_TAKEOUT
  #ifdef __linux__
      return major(sbp->st_dev) == 0;
  #endif
-
- #endif
      return 0;
  }
--- 59,68 ----
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: SOFTWARE LISTING

This section lists the software environment/source code used for the port, their purpose, and where they can be obtained.

mountd/*nfsd*:
http://carroll.aset.psu.edu/pub/linux/distributions/slackware/slackware-7.0/source/n/tcpip1/nfs-server-2.2beta46.tar.gz

This is the source code for the *mountd*/*nfsd*.

portmapper:
http://carroll.aset.psu.edu/pub/linux/distributions/slackware/slackware-7.0/source/n/tcpip1/portmap_5beta.tar.gz

The source code for the portmapper.

glibc library:

http://carroll.aset.psu.edu/pub/linux/distributions/redhat/redhat/linux/8.0/en/os/i386/SRPMS/glibc-2.2.93-5.src.rpm

This is source code for the GNU C library.

Slackware 7.0:

http://carroll.aset.psu.edu/pub/linux/distributions/slackware/slackware-7.0/

This was the first development environment used to get the *nfsd* userland software working and to get an initial understanding of how it works.

Redhat 8.0:

http://carroll.aset.psu.edu/pub/linux/distributions/redhat/redhat/linux/8.0/

This is the second development environment used after familiarity of *nfsd* was already achieved. This system became the main development environment because it more accurately mirrors the Linux environment that is provided by the XTS-400.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: ADMINISTERING NFS

A. COMPONENT COMPILATION

This section will provide instructions on how to compile the various NFS components.

Compiling the GLIBC library:

1. Go into the directory extracted from the GLIBC zip file.
2. Run “make”. If there are compilation errors, make sure to remove the temporary file “sysd-versions”.
3. The important binaries created from “make” are “libc.a”, “libc.map”, “libc.so”, and “libc_nonshared.a”. Copy these binaries into a directory where the library binaries should be stored, such as “/home/kandy/src/lib”.
4. Create a symbolic link to the dynamic library “libc.so” by running “ln –s libc.so libc.so.6”. This is necessary so that the loader can resolve dynamic library dependencies.
5. In order for a program to use the newly created library, it needs to set the environment variable LD_LIBRARY_PATH to the directory that contains the new library. This can be done by running “export LD_LIBRARY_PATH=/home/kandy/src/lib”.

Compiling the portmapper:

1. Go into the directory extracted from the portmap zip file.
2. Run “make USE_GLIBC=1 portmap”. The “USE_GLIBC=1” switches causes the portmapper to be compiled to use GLIBC.

Compiling the mountd/nfsd daemon:

1. Go into the directory extracted from the mountd/nfsd zip file.
2. Run "./BUILD". This will prompt the user with questions for how the mountd and nfsd daemons should be configured. The following answers should be used:
   - no to new inode number scheme
   - no to R/W support for multiple daemons
   - no to dynamic UID mapping
   - no to NIS UID mapping
   - 0 for UID of /etc/exports
   - 0 for GID of /etc/exports
   - no for HOST ACCESS check
   - yes to log to syslog
3. Run "make rpc.mountd rpc.nfsd".

B. COMPONENT EXECUTION

This section will provide instructions on how to run the various NFS daemons.

Running portmap:

1. Make sure the log files in the "/tmp" directory are deleted.
2. Use *sl* to change the user's security level to "min" security and "max" integrity.
3. Run *tp_edit* to move the portmap binary into a privileged directory so that it can be run in daemon mode. Use "cd" to change the current working directory to "/system". Use "add" and set the program name to "portmap", the pathname to

"/home/kandy/trans/nfs_bin/portmap", and leave the defaults for the rest of the settings.

4. Run *daemon_edit* to configure how portmap should be run. Use "add" and set the name to "portmap", the commandline to "portmap", the argument to "-d", the environment variable to "LD_LIBRARY_PATH=/home/kandy/src/lib", no for startup, sl0 for security level, il3 for integrity level, username to "network", group name to "stop", yes for High Integrity, and leave the defaults for the rest of the settings.
5. Run *start_daemon* and then type "portmap" to start the portmap daemon.

Running mountd:

1. Make sure the log files in the "/tmp" directory are deleted.
2. Use *sl* to change the user's security level to "min" security and "max" integrity.
3. Run *tp_edit* to move the mountd binary into a privileged directory so that it can be run in daemon mode. Use "cd" to change the current working directory to "/system". Use "add" and set the program name to "mountd", the pathname to "/home/kandy/trans/nfs_bin/rpc.mountd", and leave the defaults for the rest of the settings.
4. Run *daemon_edit* to configure how mountd should be run. Use "add" and set the name to "mountd", the commandline to "mountd", the argument to "-F", the environment variable to "LD_LIBRARY_PATH=/home/kandy/src/lib", no for startup, sl0 for security level, il3 for integrity level, username to "kandy", group name to "stop", yes for High Integrity, and leave the defaults for the rest of the settings.

5. Run *start_daemon* and then type “mountd” to start the mountd daemon.

Running nfsd:

1. Make sure the log files in the “/tmp” directory are deleted.
2. Use *sl* to change the user’s security level to “min” security and “max” integrity.
3. Run *tp_edit* to move the nfsd binary into a privileged directory so that it can be run in daemon mode. Use “cd” to change the current working directory to “/system”. Use “add” and set the program name to “nfsd”, the pathname to “/home/kandy/trans/nfs_bin/rpc.nfsd”, and leave the defaults for the rest of the settings.
4. Run *daemon_edit* to configure how nfsd should be run. Use “add” and set the name to “nfsd”, the commandline to “nfsd”, the argument to “-F”, the environment variable to “LD_LIBRARY_PATH=/home/kandy/src/lib”, no for startup, sl0 for security level, il3 for integrity level, username to “kandy”, group name to “stop”, yes for High Integrity, and leave the defaults for the rest of the settings.
5. Run *start_daemon* and then type “nfsd” to start the nfsd daemon.

C. SAMPLE EXPORTS FILE

```
/usr/local      192.168.0.1(ro)    192.168.0.2(ro)
/home           192.168.0.1(rw)    192.168.0.2(rw)
```

Figure 6. Example 'exports' File, From [LINU]

The first field is the name of the directory that will be shared. The fields following that is a list of host that are allowed to use the shared directory along with the type of access allowed specified in parentheses. 'ro' will allow a host to have read-only access, while 'rw' will allow the host to have both read and write access.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[BARR] Barr, T., "Linux NFS-HOWTO," August 25, 2002, http://nfs.sourceforge.net/nfs-howto/index.html, March 15, 2004.

[BELL73] Bell, D., and LaPadula, L., "Secure Computer Systems: Mathematical Foundations and Model," *MITRE Report*, MTR 2547, Vol. 2, November 1973.

[BIBA77] Biba, K., "Integrity Considerations for Secure Computer Systems," ESD-TR-76-372, ESD/AFSC, Hanscom AFB, Bedford, MA, April 1977.

[COUL] Coulouris, G., Dollimore, J., and Kindberg, T., "Distributed Systems: Concepts and Design," Third Edition, Addison-Wesley, 2001.

[FOC] Focke, M., "XTS-400 Trusted Computer System Technical Overview," October 29, 2003, http://www.digitalnet.com/solutions/info_sec_sol/pdf/XTS-400TechnicalOverview.pdf, January 13, 2004.

[GEM] National Security Agency, National Computer Security Center, “Final Evaluation Report, Gemini Trusted Network Processor,” CSC-EPL-94/008, September 6, 1994, http://www.radium.ncsc.mil/tpep/epl/entries/CSC-EPL-94-008.html, January 10, 2004.

[IRV04] Irvine, C. E., Levin, T. E., Nguyen T. D., Shifflett, D., Khosalim, J., Clark, P. C., Wong, A., Afinidad, F., Bibighaus, D., and Sears, J., "Overview of a High Assurance Architecture for Distributed Multilevel Security,” Submitted for publication, 2004.

[MANU] DigitalNet, "XTS-400 User’s Manual," XTDOC0005-02, Herndon, VA. January 2003.

[MCKU] McKusick, M., Bostic, K., Karels, M., and Quarterman, J., "The Design And Implementation of The 4.4 BSD Operating System," Addison-Wesley, 1996.

[PEMB] Pemberton, S., "The Ergonomics of Software Porting," February 24, 1999, http://ftp.cwi.nl/steven/enquire/enquire.html, February 10, 2004.

[PERRINE] Perrine, T., "The Kernelized Secure Operating System (KSOS)," *login*, Vol. 27, No. 6, December 2002, www.usenix.org/publications/login/ 2002-12/pdfs/perrine.pdf, February 15, 2004.

[RED] Red Hat, "Red Hat Support", December 10, 2003, https://www.redhat.com/docs/manuals/linux/, January 27, 2004.

[SAN85] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. "The Design and Implementation of the Sun Network File System," In Proceedings Usenix Summer Conference, pages 119-130, June 1985.

[SALT75] Saltzer, J. H., Schroeder, M. D., "The Protection of Information in Computer Systems," Proceedings of the IEEE, 63(9):1278-1308, September 1975.

[SCHELL74] Karger, P., and Schell, R., "Multics Security Evaluation: Vulnerability Analysis," ESD-TR-74-193 Vol. II, ESD/AFSC, Hanscom AFB, Bedford, MA, June 1974.

[SPEC89] Sun Microsystems, "NFS: Network File System Protocol specification," RFC 1094, March 1989, http://www.faqs.org/rfcs/rfc1094.html, February 2, 2004.

[TP85] DoD 5200.28 STD, Department of Defense Standard, "Department of Defense Trusted Computer System Evaluation Criteria," December 1985.

[WANG99] Final Evaluation Report Wang XTS-300. April 27, 1999, http://www.radium.ncsc.mil/tpep/library/fers/CSC-EPL-92-003-C.ps.gz, February 4, 2004.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. George Bieber
   OSD
   Washington, DC

4. RADM Joseph Burns
   Fort George Meade, MD

5. Deborah Cooper
   DC Associates, LLC
   Roslyn, VA

6. CDR Daniel L. Currie
   PMW 161
   San Diego, CA

7. LCDR James Downey
   NAVSEA
   Washington, DC

8. Richard Hale
   DISA
   Falls Church, VA

9. LCDR Scott D. Heller
   SPAWAR
   San Diego, CA

10. Wiley Jones
    OSD
    Washington, DC

11. Russell Jones
    N641
    Arlington, VA

12. David Ladd
Microsoft Corporation
Redmond, WA

13. Dr. Carl Landwehr
National Science Foundation
Arlington, VA

14. Steve LaFountain
NSA
Fort Meade, MD

15. Dr. Greg Larson
IDA
Alexandria, VA

16. Ray A. Letteer
Head, Information Assurance, HQMC C4 Directorate
Washington, DC

17. Penny Lehtola
NSA
Fort Meade, MD

18. Ernest Lucier
Federal Aviation Administration
Washington, DC

19. CAPT Sheila McCoy
Headquarters U.S. Navy
Arlington, VA

20. Dr. Ernest McDuffie
National Science Foundation
Arlington, VA

21. Dr. Vic Maconachy
NSA
Fort Meade, MD

22. Doug Maughan
Department of Homeland Security
Washington, DC

23. Dr. John Monastra
Aerospace Corporation
Chantilly, VA

24. John Mildner
SPAWAR
Charleston, SC

25. Marshall Potter
Federal Aviation Administration
Washington, DC

26. Dr. Roger R. Schell
Aesec
Pacific Grove, CA

27. Keith Schwalm
Good Harbor Consulting, LLC
Washington, DC

28. Dr. Ralph Wachter
ONR
Arlington, VA

29. David Wirth
N641
Arlington, VA

30. Daniel Wolf
NSA
Fort Meade, MD

31. CAPT Robert Zellmann
CNO Staff N614
Arlington, VA
Robert.Zellmann@navy.mil

32. Dr. Cynthia E. Irvine
Naval Postgraduate School
Monterey, CA

33. David Shifflett
Naval Postgraduate School
Monterey, CA

34. Kandy Phan
SFS students: Civilian, Naval Postgraduate School
Monterey, CA