1999-03

Secure local area network services for a high assurance multilevel network

BryerJoyner, Susan.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

SECURE LOCAL AREA NETWORK SERVICES FOR A HIGH ASSURANCE MULTILEVEL NETWORK

by

Susan BryerJoyner and Scott D. Heller

March 1999

Thesis Advisor:	Cynthia E. Irvine
Second Readers:	James P. Anderson
	James Bret Michael

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1999	Master's Thesis

4. TITLE AND SUBTITLE SECURE LOCAL AREA NETWORK SERVICES FOR A HIGH ASSURANCE MULTILEVEL NETWORK

5. FUNDING NUMBERS

6. AUTHOR(S) Susan BryerJoyner and Scott D. Heller

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)

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)*

To reduce the cost and complexity of the current DoD information infrastructure, a Multilevel Secure (MLS) network solution eliminating hardware redundancies is required. Implementing a high assurance MLS LAN requires the ability to extend a trusted path over a TCP/IP network. No high assurance network trusted path mechanisms currently exist.

We present a design and proof-of-concept implementation for a Secure LAN Server that provides the trusted path between a trusted computing base extension (TCBE) servicing a COTS PC and protocol servers executing at single sensitivity levels on the XTS-300. The trusted path establishes high assurance communications (over a TCP/IP network) between a TCBE and the Secure LAN Server. This trusted channel is used first for user authentication, then as a trusted relay between the protocol server and TCBE. All transmitted data passed over the LAN can be protected by encryption, providing assurance of integrity and confidentiality for the data.

This thesis documents the implementation of a demonstration prototype Secure LAN Server using existing technology, including high assurance systems, COTS hardware, and COTS software, to provide access to multilevel data in a user-friendly environment. Our accomplishment is crucial to the development of a full scale MLS LAN.

14. SUBJECT TERMS multilevel security, trusted path, secure session, high assurance server, thin client			15. NUMBER OF PAGES 236
			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. 239-18 298-102

Approved for public release; distribution is unlimited.

SECURE LOCAL AREA NETWORK SERVICES FOR A HIGH ASSURANCE MULTILEVEL NETWORK

Susan BryerJoyner
Lieutenant, United States Navy
B.S., Rensselaer Polytechnic Institute, 1991

Scott D. Heller
Lieutenant, United States Navy
B.A., University of Rochester

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1999

Authors:

[redacted]
Susan BryerJoyner

[redacted]
Scott D. Heller

Approved by:

[redacted]
Cynthia E. Irvine, Thesis Advisor

[redacted]
James P. Anderson, Thesis Second Reader

[redacted]
James Bret Michael, Thesis Second Reader

[redacted]
Dan Boger, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

To reduce the cost and complexity of the current DoD information infrastructure, a Multilevel Secure (MLS) network solution eliminating hardware redundancies is required. Implementing a high assurance MLS LAN requires the ability to extend a trusted path over a TCP/IP network. No high assurance network trusted path mechanisms currently exist.

We present a design and proof-of-concept implementation for a Secure LAN Server that provides the trusted path between a trusted computing base extension (TCBE) servicing a COTS PC and protocol servers executing at single sensitivity levels on the XTS-300. The trusted path establishes high assurance communications (over a TCP/IP network) between a TCBE and the Secure LAN Server. This trusted channel is used first for user authentication, then as a trusted relay between the protocol server and TCBE. All transmitted data passed over the LAN can be protected by encryption, providing assurance of integrity and confidentiality for the data.

This thesis documents the implementation of a demonstration prototype Secure LAN Server using existing technology, including high assurance systems, COTS hardware, and COTS software, to provide access to multilevel data in a user-friendly environment. Our accomplishment is crucial to the development of a full scale MLS LAN.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENT

We are very grateful for and impressed with the quality of input and support we have received during our efforts to conduct this research. First and foremost, the knowledge and efforts of Dr. Cynthia E. Irvine, our thesis advisor, made the successful development of this product possible. It is Dr. Irvine's and James P. Anderson's concept of a MLS LAN that initially sparked our interest, and their encouragement kept our motivation high throughout the entire process. We appreciate the guidance, support and enthusiasm provided by James P. Anderson and James Bret Micheal. Many others played integral roles in facilitating this work and deserve special recognition. Paul Clark provided experience with the STOP operating system and expertise in maintaining the Computer Security Lab LAN. Wang Government Services management and technical support provided key answers to varied, and frequently urgent, XTS-300 capability and interface questions. Supporters from Wang Government Services include George Webber, Bob Wherley, Paul Barbieri, John Ata, and Hans Anderson. Without their patience and help, the implementation demonstration would not have been possible.

I. INTRODUCTION

A. BACKGROUND

Information control is pervasive in all aspects of our lives. Business secrets, financial statements, what your children are allowed to see on TV, personal diaries, military plans, and political agendas are all examples of information that needs to be controlled. Diaries disclosed can lead to personal embarrassment, political agendas revealed can ruin a career, and military plans leaked may kill people or even cause the downfall of a society. Clearly, the control of information is vitally important to all of us.

A security policy dictates who has access to which information and the procedures for accessing information. A well thought out security policy is essential in any organization where controlled sharing is required. In a computer, controlled sharing occurs when people are able to access the system where all of the information is stored, but a person's ability to access certain information is determined by his permissions. The first step in protecting our information resources is to delineate what rules and procedures users must follow in order to access information. Collectively, these rules and procedures compose a security policy.

In the past, security policies were primarily directed towards the protection of paper documents including text, photos, and microfiche. The policies focused on controlling access and limiting duplication of the protected documents. In the last 30-40 years, there has been a significant change. Most information is no longer stored in hard copy form, but is entrusted to some electronic medium. The benefits of using digital data, such as search speed, minimal volume, ease of reproduction, and ease of modification, have all combined to accelerate the use of electronic storage via computers.

The move towards electronic storage and management of information changed the methods used to implement security policies, but the amount of money dedicated to security remains limited. When organizations must decide where to concentrate their fiscal resources

in order to protect their information decisions are generally based on the cost-effectiveness of the security solution. Many types of security relate to computers and the information they process and store. Some of the security areas applied to the protection of hard copy documents and are well understood and implemented: physical, personnel, emanation, and operations security. The Department of Defense (DoD) has spent years and significant effort to improve security in these fields; additional gains cannot be achieved without a disproportionate increase in expense. Computer security, a relatively new field that addresses the different concerns associated with computers, offers the most cost-effective avenue for improvement.

Computer security focuses on the enforcement of an information security policy in the electronic realm. In this area, the threat of the inside user who decides to steal information has been joined by that of an outside attacker who is able to create, and introduce to the target system, a malicious application. The application may steal many documents very quickly or modify existing data in such a way that it is no longer useful or is actually detrimental to their intended use. Over the past 30 years, many researchers have conducted extensive research on countering the effects of malicious code and other significant computer security threats. However, much of the work has gone unheeded and unused due to the tremendous expense involved with purchasing and using the systems that were developed to provide the solutions. The focus of this paper is on using existing secure components to enforce a given policy, while ensuring that the cost of implementation is not prohibitive and the ease of use expected by today's users is maintained. We intend to provide the building blocks for a secure local area network (LAN) that is economical, easy to use, and based on a security architecture comprised of both high assurance and commercial-off-the-shelf (COTS) components.

A secure system is one that accurately implements a specified security policy [Ref. 1]. The degree of assurance that its security features and architecture correctly enforce security policy can be measured against well-understood criteria such as the "Department of Defense Trusted Computer System Evaluation Criteria" (TCSEC) [Ref. 2]. The TCSEC

offers criteria for evaluating systems with a range of security features. In the past, many systems that met the highest levels of assurance also proved to be the least user friendly. High assurance systems, that meet the TCSEC Class B3 requirements, have often been dedicated to specialized tasks with a limited number of expert users; consequently, the lack of a user-friendly interface was often not an issue. As more enterprises, including the military, move toward the use of COTS equipment there is a greater desire to have high assurance servers that can provide controlled sharing of information within a LAN and still retain compatibility with COTS products. High assurance multilevel workstations can provide controlled sharing, but are expensive, difficult to use, and incompatible with COTS software.

Conflicting mandates such as minimizing cost while maximizing functionality have rendered expensive, high assurance workstations infeasible. Fortunately, technological advances in networking have revived the centralized host approach. Originally, the mainframe computers that provided storage and computing power were exceedingly expensive. To use mainframe computers more efficiently, inexpensive "dumb" terminals were connected to a central server providing multiple users with concurrent access to the resources, thus forming a network. Technological advancements led to the creation of an affordable desktop computer. These personal computers (PCs) became cheaper to purchase and install for organizations than one centralized server coupled with many terminals. Consequently, networks moved away from centralized topologies in the late 1980's and 1990's.

The power of today's computers and the cost of installing licensed software on each PC have revived the idea of utilizing a centralized server to provide network services to thin clients[1] or inexpensive personal computers. Centralized hosts are capable of providing a variety of services that extend the functionality of inexpensive PCs connected to LANs;

---

1 Thin clients, also known as network computers, are computers with minimal processing and possibly no permanent storage capabilities; they depend on a server to process and store data, and provide a user interface to the client.

examples of these services include electronic email, database management, fileservers, and directory services. The protection of this consolidated data storage lends itself to the use of one high assurance server. A network with a high assurance server and inexpensive PCs or thin clients as terminals has the potential to be less expensive than a network composed of highly capable PCs with individually licensed software suites. A centralized approach has the potential to provide economically sound secure networks.

Proof that modern enterprises are seriously considering the centralized approach is evident in the military. Specifically, the US Navy is exploring the concept of a Navy Virtual Intranet (NVI) which will electronically interconnect and provide information services to the Navy and Marine forces, and civilian employees afloat and ashore. The proposed functional architecture is based on commercial-off-the-shelf (COTS) hardware and software with security implemented at each level of the architecture. One of the basic premises of the plan is that operational and financial constraints will preclude the Navy from absolute assurance that the threat will be kept out of the information systems during future conflicts [Ref. 3]. We believe that it is possible to achieve a higher level of assurance than is available from COTS products without sacrificing the requisite interoperability.

Currently, the Navy and other services segregate multiple security levels by providing independent network infrastructures for each level of information needed. Consequently, users who need access to networks at three classification levels will have three redundant PCs on their desk. Besides the waste of resources that is immediately apparent, there is an inherent security vulnerability in this setup; there are no labels associated with the information in its electronic form, and possibly not in its hard copy form. The NVI retains this existing network structure. We believe that research associated with the MLS LAN project, including this thesis, will demonstrate a high assurance system architecture that retains the interoperability with COTS hardware and software needed by the Navy and other organizations in DoD and US government.

B. GOALS OF THE THESIS

As outlined above, a MLS LAN implementation must overcome three basic obstacles: removing redundant PCs at the user's desk, resolving incompatibility issues between high assurance platforms and COTS software and hardware, and mitigating the high cost of high assurance platforms. To solve the MLS LAN problem, we will leverage existing technology, including high assurance systems, COTS hardware, and COTS software, to provide access to multilevel data in a user-friendly environment. High assurance multilevel platforms that permit controlled sharing of sensitive information by users at multiple security levels exist. While high assurance platforms are prohibitively expensive to put on each desktop, they are excellent for use as a server in a centralized network configuration. An ideal solution would permit use of COTS software to manipulate data that is stored on a high assurance multilevel server. A high assurance multilevel platform that implements the Bell-LaPadula Model[2], such as the Wang XTS-300, is crucial to ensuring that the correct subject/object dominance relation between security levels is enforced. When used as a server, the security features present in the XTS-300 can mediate access to stored data. Designing a network that can securely distribute information at multiple classification levels to inexpensive single-level workstations will allow the DoD to conserve resources by eliminating redundant desktop computer systems and networks.

---

2 "A formal state transition model of computer security that describes a set of access control rules. In this formal model, the entities in a computer system are divided into abstract sets of subjects and objects. The notion of a secure state is defined and it is proven that each state transition preserves security by moving from secure state to secure state; thus, inductively proving that the system is secure. A system state is defined to be "secure" if the only permitted access modes of subjects to objects are in accordance with a specific security policy. In order to determine whether or not a specific access mode is allowed, the clearance of a subject is compared to the classification of the object and a determination is made as to whether the subject is authorized for the specific access mode. The clearance/classification scheme is expressed in terms of a lattice." The model also defines the Simple Security Condition to control granting a subject read access to a specific object, and the *-Property to control granting a subject write access to a specific object. [Ref. 2]

Additionally, this architecture will be able to support rapid upgrades of commercially developed office productivity products at the workstation without requiring modifications to the trusted components [Ref. 4]. This will be possible if the PC workstation and the software on the PC are not required to be trusted. A network administrator can then simply upgrade the software on the user's PC and allow continued operation.

The objective of the secure LAN development project is to utilize a LAN with a high assurance server and COTS workstations to provide a secure processing environment in which user functions or programs can be securely integrated at virtually any time while still preserving the security of existing data. Our proposed solution involves networking COTS PCs equipped with a trusted computing base extension (TCBE) to the existing trusted computing base (TCB) on the XTS-300 (see Figure 1) [Ref. 4]. The TCBE negotiates a trusted path across the network with XTS-300 when it sends the secure attention sequence (SAS). The user at the PC, communicating with the TCBE, is then able to use the trusted path to initiate a secure session on the XTS-300.

This thesis develops and demonstrates a procedure for establishing a trusted path and secure session between a thin client and a high assurance multilevel server over an untrusted LAN. It involves a rigorous software engineering approach applied to the design, implementation, and analysis of our procedure for establishing a trusted path and a secure session. The goal of this research is to provide the foundation upon which a secure multilevel LAN can be created using a single multilevel server and numerous, inexpensive thin clients. Initially the thin clients will be COTS PCs modified to operate as write-less clients.

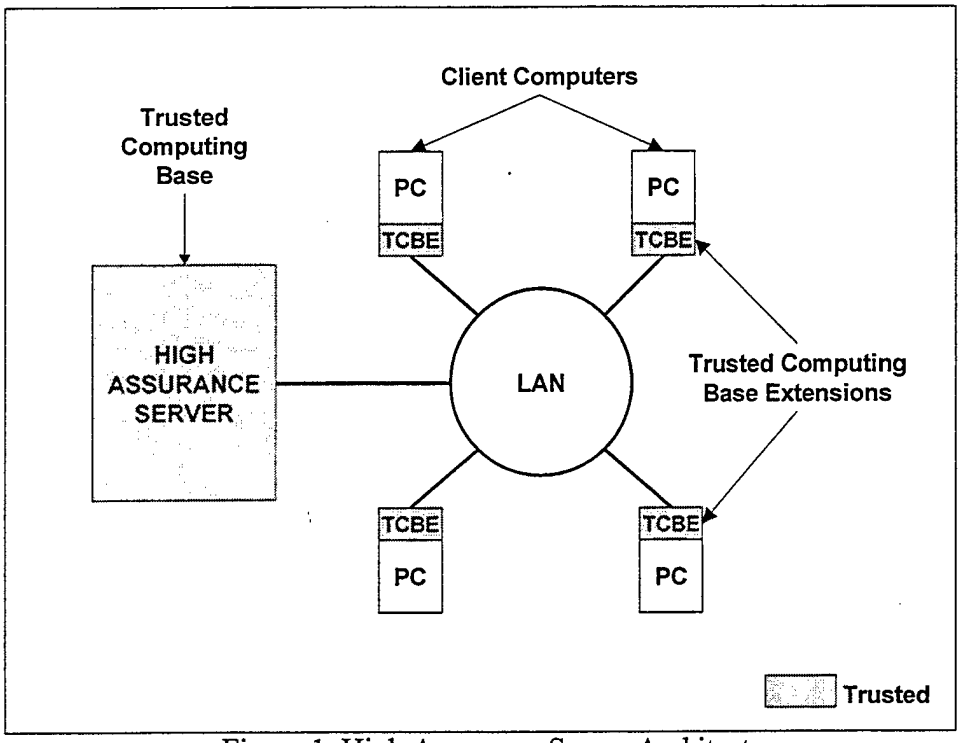

Figure 1. High Assurance Server Architecture

C. JUSTIFICATION FOR A TRUSTED PATH

1. Why does our design require a Trusted Path?

> The TCB shall support a trusted communication path between itself and users for use when a positive TCB-to-user connection is required (e.g., login, change subject sensitivity level). Communications via this trusted path shall be activated exclusively by a user or the TCB and shall be logically and unmistakably distinguishable from other paths. [Ref. 2: p. 107]

A trusted path, as mandated above in the TCSEC, is intended to provide a guaranteed conduit for information exchange between the TCB and user (see Figure 2). The trusted path must ensure both ends of the connection cannot be spoofed and that all messages are tamperproof. This means that when the user initiates a connection to the server, he/she is guaranteed to be communicating with the TCB and with no other process. From the server's perspective, the server must be assured that it is communicating with a

process executing on a piece of equipment that can be uniquely and positively identified and provides a conduit to the user that cannot be subverted.

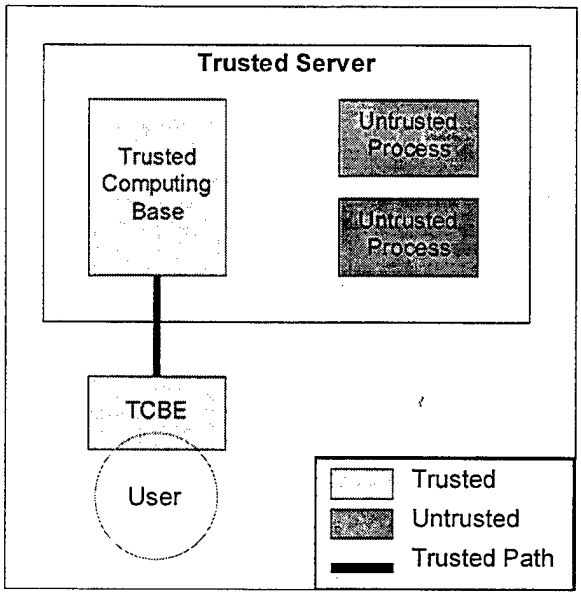

Figure 2. Trusted Path

These standards for secure communication via a trusted path are very stringent, but necessary since the trusted path is a building block of a protected session. Without these guarantees, it is not possible to assume any subsequent communication between the server and the various clients can be protected. The trusted path is used to perform user identification and authentication, negotiate session levels, and possibly invoke trusted subjects[3] on the server to execute on behalf of the remote user.

If we step back a bit and consider two individuals who wish to have a sensitive conversation, the concept of a trusted path may become a bit clearer. They have not met before, but have previously arranged to meet over the telephone. Upon meeting, the first question on each of their minds will be "How do I know I am talking to the right person?" After the first question is resolved, the second question will be "How do I know this

---

3 A trusted subject is a subject that is part of the TCB. Within a specified range of access classes it is not constrained by the confinement property, but is trusted not to actually do so. It is required for special operations that span access classes; an example of such an operation is downgrading information.

conversation is private?" Once these two questions have been answered satisfactorily, the conversation can proceed with some level of confidence that it is appropriate and private.

Using a trusted path to initiate electronic communication is the method of choice for answering the two preceding questions in a networked environment. Extending this to a physically unprotected LAN is the primary goal of this research. Our trusted path will use public-key cryptography to authenticate the TCBE to the server, and vise versa, when the user initiates the trusted path.

During a session, there are two possible communication modes available to the user: trusted path or normal. A trusted path is required for direct communication between the user and the TCB, such as during user authentication. The encryption that protects trusted path communication depends on a one-time session key that is established when the user initiates a session. This session key is negotiated by the trusted path server and the TCBE during the initial trusted path setup using Oakley[4] or some other equally strong[5] key exchange algorithm.

Normal communication occurs when the user does not have to communicate directly with the TCB. Once the session is established and the user begins working, normal communication may or may not need to be encrypted. The option to not encrypt non-trusted path data will be available if data protection is not required.

The design and implementation of a trusted path mechanism will solve the problem of hardware authentication between the server and the TCBE for initial identification and authentication functions and trusted commands. Additionally, the trusted path will provide secure, tamperproof, communication between the server and the TCBE over a physically unprotected LAN. The trusted path mechanism will not prevent denial of service attacks,

---

4 The Oakley Key Determination Protocol uses a hybrid Diffie-Hellman technique to establish session keys on Internet hosts and routers. Oakley provides the important security property Perfect Forward Secrecy and is based on cryptographic techniques that have survived substantial public scrutiny. [Ref. 5]

5 In cryptography the word strong has special meaning; an algorithm is considered strong if there are no known methods exist to break the crypto-system with existing technology and knowledge. [Ref. 6]

but, if designed carefully, may limit the effectiveness of some denial of service attacks by requiring proper public-key authentication before initiating a resource intensive key exchange.

To summarize the benefits of a trusted path in our design, it will allow the server and the TCBE to authenticate each other and then enable secure communication between the server and the TCBE. Following the creation of the trusted path, the user on the client side of the TCBE will be allowed to login and use the services provided by the secure server. The trusted path does not address the issues surrounding communication between the TCBE and the COTS client. What it will provide is a mechanism for extending the trusted computing base to multiple TCBEs over a physically unprotected LAN using TCP/IP. This approach should be readily extensible to a wide range of applications.

2. What can happen if we do not establish a trusted path?

Without a trusted path, the network cannot be certified at Class B3 in accordance with the Trusted Network Interpretation (TNI) of the TCSEC [Ref. 7], which is a primary goal of our project. Since a trusted path is required by the TCSEC for a Class B3 certification, the trusted path functionality is not optional. To understand the TCSEC requirement consider the possibility that there exists an untrusted process with malicious intent. Now examine the remote login procedure and the fact that without a trusted path to protect communication, the malicious application can listen to all traffic since there is no trusted path protecting the communications. When the remote user attempts to login, the malicious application could listen for the login message and possibly impersonate the server or record the login for a replay attack later. The author of the malicious code can replay the login data since, without the trusted path, the hardware initiating the communication is not identified. Now that this application can login as an authorized user, this process could modify, reroute, or examine any traffic that the malicious programmer was creative enough to anticipate. A trusted path protects communication between a user and a TCB, such as login and session negotiation, from interference or replay by untrusted code since the hardware at both endpoints have been properly authenticated.

D. ESTABLISHING A TRUSTED PATH

1. What are the design requirements of a trusted path?

These requirements have been touched upon above, but we will restate them here. First and foremost, the trusted path must guarantee the ability to authenticate the identity of each party involved in the creating the communication session. In our design, outlined in Chapter II of this thesis, these parties will be the Trusted Path Server and the TCBE, which provide an interface through which the user communicates to the TCB. The Trusted Path Server will negotiate trusted paths with the various TCBE clients. The idea is to establish a secure conduit between the Trusted Path Server and the TCBE utilizing public-key encryption and signatures, then to create a one-time session key to protect the trusted path communications. This one-time session key may, or may not, be used for the follow-on secure session communications. Establishing the trusted path will only guarantee communication security between the TCBE hardware and the Trusted Path Server on the XTS-300 for use during the hardware and user identification and authentication states. It is important to remember that the trusted path is designed to authenticate the hardware, not the user attempting to use the TCBE client. User authentication information will be exchanged in a secure manner using the one time key generated during the negotiation of the trusted path and then forwarded to the existing STOP[6] system calls for user identification and authentication.

The second function of the trusted path is to guarantee all communication between the Trusted Path Server and the TCBE is tamperproof. In other words, we must be assured

---

6 STOP is a multilevel secure operating system developed and supported by Wang Government Services, Inc. STOP consists of four components: the Security Kernel, which operates in the most privileged ring and provides all mandatory, subtype, and a portion of the discretionary, access control; the TCB System Services, which operate in the next-most-privileged ring, and implements a hierarchical file system, supports user I/O, and implements the remaining discretionary access control; Trusted Software, which provides the remaining security services and user commands; and Commodity Application System Services (CASS), which operate in a less privileged ring and provide the UNIX-like interface. CASS is not in the TCB. [Ref. 8]

that a third party cannot manipulate messages such that the meaning of a message is changed. This goal can be realized using symmetric key encryption. In addition, encryption algorithm can be used to provide data communication integrity, i.e. to ensure that if one bit changes in the encrypted text there will be a large change in the decrypted message. With a good algorithm, the change will be large enough so that the decrypted message will not decrypt properly. This property is called diffusion [Ref. 9: p. 60]. Diffusion in cryptography means that when one character in the input data changes the cipher text output changes dramatically. The reverse of this implies that if a third party attempts to change the cipher text to manipulate the plain text the decrypted message is gibberish.

Ensuring that the cipher text cannot be manipulated in a useful manner is only one element of making the message traffic tamperproof. We must also ensure that the message cannot be substituted wholesale. In other words we must ensure no other person or process can intercept a message and substitute their own message which could be assumed to be from a legitimate party. This attack, known as spoofing, can be prevented by protecting the symmetric encryption key and by selecting a suitable encryption algorithm. The suitability of an encryption algorithm depends on requirements established by the cognizant authority. Possible requirements include the length of time that the information must be protected and assumptions about adversary resources and expertise. The first step towards protecting the symmetric encryption key is to never pass the key in the clear. Ideally, this key should never be transmitted using the same medium as the future traffic encrypted by the symmetric key. In order to prevent the key from ever being vulnerable, a strong high assurance public-key exchange algorithm must be used. In this manner both parties can calculate the symmetric key and have no need to transmit the key data over the LAN that will carry the encrypted text. Selection of the key exchange algorithm and the symmetric encryption algorithm will be the topics of future research, but for now let us assume that suitable algorithms exist and can be implemented in our chosen architecture.

If these two requirements are met, both parties will know exactly who they are communicating with and all messages between the parties will be guaranteed to be as

intended by each party. In summary, the trusted path will provide hardware authentication and assurances that the messages will be confidential and tamperproof. A secure LAN can be built from this as a building block.

E. JUSTIFICATION FOR A SECURE SESSION

The trusted path is instrumental in the establishment of a secure session. Without the ability to communicate directly with the trusted computing base (TCB), a remote user would not be able to provide the information that is required to login, set session level, and maintain accountability in the system with the requisite level of assurance. What exactly is a secure session? A session refers to the connection between the client computer and the remote server. A secure session must be established and maintained in a manner that preserves a secure state on the remote server. Consequently, the user must provide information to the TCB that facilitates the enforcement of the security policy before being able to establish the session. In order to maintain a session in a secure manner, its transmissions must have some characteristic that thwarts any attempt at imitation and prevents useful interception by untrusted processes. The cryptographic algorithms described later in this paper provide this characteristic.

F. ESTABLISHING A SECURE SESSION

A secure server must accurately enforce the security policy for all attempts to access its information, whether the attempts are from local or remote users. The user must provide information directly to the TCB so the server can control access to the information and provide user accountability. The degree of confidence that the server is correctly implementing the security policy is its level of assurance and is very important.

1. Accountability

> All personnel of the Department of Defense are personally and individually responsible for providing proper protection to classified information under their custody and control. [Ref. 11]

Accountability provides a means of determining the responsible party in a given situation and has two requirements.

1. Each subject and object in the system must be uniquely identified in order to track actions with the requisite granularity.
2. The actions must be recorded and protected from modification.

The military, where the actions of an individual can endanger national security or lives, has always recognized the importance of accountability and implemented it where necessary. Commercial ventures are also becoming increasingly aware of the inherent value of the information stored on their computers. Consequently, security administrators must be able to track every successful and failed attempt to access protected information in order to identify and discipline users who act inappropriately. In order for a system to provide accountability, it must implement identification and audit.

a. Identification and Authentication

Identification is "...the process that enables recognition of an entity by a system, generally by the use of unique machine-readable user names" [Ref. 12]. Identification without authentication, however, is not useful to systems that are trying to provide accountability. Authentication is the "means of establishing the validity of" the identity. [Ref. 13] There are three accepted methods of authentication that can be used alone or in any combination:

1. Something the user knows (a password or Personal Identification Number (PIN));
2. Something the user has (a token or smart card);
3. Something the user is (a unique biological trait such as a fingerprint or retina pattern).

While there are benefits and drawbacks for each of these methods, the most common authentication method being implemented currently is the use of passwords. Since this is the case for the XTS-300, this section will briefly explain the advantages and concerns associated with the use of passwords.

The strength of a password derives from two sources, its composition and its secrecy. If one of these two components is flawed, a password cannot be trusted to provide adequate security. The perfect password would have a different composition every time it is used, a one-time password. Although the use of one-time passwords is possible, it usually involves adding hardware and software to the existing system. The XTS-300 does not implement one-time passwords, so they will not be discussed. Passwords used more than once can be relatively secure against most forms of attack if users follow certain composition rules while changing their passwords periodically. These rules are outlined in the DoD Password Management Guideline [Ref. 14].

However securely a password is composed, it cannot provide security and be used for authentication if it is known by more than one person. Therefore, it is very important that each user protects his personal password in the following ways: do not write it down, do not share it with anyone, and do not let anyone see you type it. Unfortunately, users have limited control over their passwords when they are transmitted over a network. System administrators should ensure that there is adequate security in place to protect passwords from electronic monitoring. They should also protect the password files on the network server(s) from unauthorized access or modification.

Access control to both the system and the information contained on the system is very important. The identification and authentication process is a crucial component of computer security since it provides the basis for most types of access control and for establishing user accountability. Systems evaluated at the Class B3 level use the clearance and authorizations associated with the user to properly mediate access to objects. In order to meet Department of Defense security requirements, we want to protect information and provide user accountability while still allowing authorized users access to information.

> Classified information and sensitive unclassified information shall be safeguarded at all times while in AISs. Safeguards shall be applied so that such information is accessed only by authorized persons, is used only for its intended purpose, retains its content integrity, and is marked properly as required. [Ref. 15]

b. Audit

Audit is the final component of the accountability mechanism. Being able to identify what actions a subject performs on an object is of limited usefulness unless the system records the actions and protects that record from modification. The security policy that must be enforced determines the granularity of the audit trail. The system administrator must choose auditable actions carefully, especially if storage space is a problem, because audit trails can grow rapidly.

2. Assurance

Brinkley and Schell use a library analogy[7] to illustrate one of the basic concepts of computer security: assurance.

> Of course, we must not only have a good security system; we must also implement it correctly. If a guard is subject to subversion or if our vault has walls of paper rather than steel, the security we provide will not be very effective. [Ref. 16]

Assurance is "a measure of confidence that the security features and architecture of an AIS accurately mediate and enforce the security policy" [Ref. 15]. In the library analogy, we have more confidence that vault walls will allow access only through legitimate entrances than we do that regular room walls could provide such a level of protection. The walls of a vault are built to be secure against penetration. The walls of a normal room are generally intended to provide separation and might have weak spots such as windows that are vulnerable to penetration. A vault inherently has a higher

---

[7] The authors draw parallels between the protection mechanisms used in protecting sensitive hard copy documents and those that should be implemented to provide security for information stored in a computer. For example, when trying to enforce access control, the first step in protecting hard copy documents might be locking them up in a vault. However, the documents still have to be used so a method of allowing authorized people to access them must be created. If a door is installed, we must then provide some set of controls over who can enter. Posting a guard at the door and providing a list of who is authorized access establishes a method of controlling who may enter the vault. The analogy continues building upon itself to explore other computer security concepts such as authentication, an audit trail, assurance in implementation, etc.

assurance of physical security than a normal room, even one with added features such as locks and barred windows, because it is built to meet certain security specifications.

Similarly, computer security requirements can be fulfilled by using either trusted systems or add-on components that provide security features. Relating these concepts to the library analogy, a vault would represent the trusted system and add-on security components could be bars on the window of a normal room. The use of add-on components (see Figure 3) is the easiest and cheapest method because it does not require any modification of the system development phase, but can be incorporated after the system has been built. There are two inherent vulnerabilities in this configuration. The first is that if the add-on security component can be circumvented, the system is vulnerable. In the library analogy, if an intruder penetrates the walls, he can gain access to the room. Even if the add-on security component cannot be by-passed, the user cannot have much confidence that the system will behave in the manner expected because the development phase contained no controls. In terms of our library analogy, even if the guard prevents unauthorized access through the door and the walls cannot be penetrated, there is no guarantee that the bars cannot be defeated.

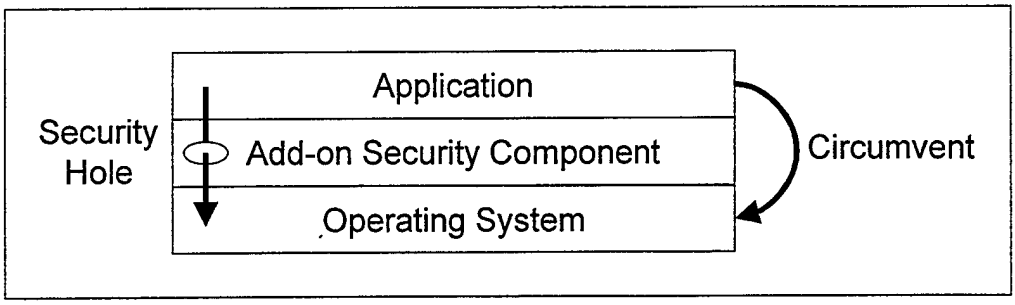

Figure 3. Add-on Security Component

On the other hand, a vault is built from the ground up to meet certain security criteria and therefore has a higher level of assurance. The same logic applies to the development of trusted systems. In this paper we use the original definition of "trust" where it is used to describe the "level of confidence that a computer system will behave as expected" [Ref. 17]. A trusted system is developed in accordance with specified security and assurance requirements [Ref. 2].

G. TERMINOLOGY

The generally accepted definition of a trusted subject is a subject that has a range, where the read class strictly dominates the write class, within which the subject is not constrained by the confinement property (*-property). Normally, a trusted subject is distinctly different from a privileged subject. A privileged subject is one that has privileges to write to privileged (i.e., protected) data structures.

The XTS-300 combines the discrete properties described above and assigns them to what it also calls a trusted subject. "A process is trusted on the XTS-300 (a trusted subject) either if the process' integrity level allows manipulation of TCB databases (an integrity level of at least operator) or if the process possesses privileges that exempt it from specific access control rules (for example, the privilege to be exempt from the security *-property)." [Ref. 18]

In order to distinguish between the two subsets of the XTS-300 trusted subject, we call a subject whose integrity level allows manipulation of TCB databases a *privileged subject*. A subject with privileges that exempt it from specific access control rules is called a *trusted subject*. It is important to note that *privileged subjects* are not always *trusted subjects*, but every *trusted subject* is a *privileged subject* in the XTS-300.

A program that is installed with a maximum integrity level of at least operator can be assigned a privilege set that potentially exempts it from specific access control rules during execution. Normally, a *privileged subject* invokes those privileges only at the specific times they are required, becoming a *trusted subject* temporarily. After the *trusted subject* has performed the desired action, the privileges are revoked, and the *privileged subject* resumes execution.

H. ORGANIZATION OF THESIS

Chapter II provides information on the software module design process. Chapter III presents the final design and discusses the implementation phases. Chapter IV introduces

other products intended to provide similar multilevel security (MLS) functionality, then compares and contrasts those products with the MLS LAN project and presents the conclusions. The design documents for the Secure LAN Server are included as Appendices A and B. The source code is included in Appendices C, D, and E. Appendix F contains a glossary of terms and acronyms used in this paper.

II. SOFTWARE MODULE DESIGN PROCESS

A. INTRODUCTION

While the implementation of protocol servers on a high assurance platform is intended to improve security, another goal of this thesis was to have minimal impact on the end user. Consequently, the design attempted to replicate existing user interfaces to the greatest extent possible while minimizing additional user interaction.

- The hardware authentication required for establishing the trusted path occurs automatically when the user depresses the SAK at the TCBE and is transparent to the user.
- User authentication, which must occur before a session can be established on the server, requires user input but the procedure resembles the login process normally associated with accessing the XTS-300.
- Setting the session level is automatic if the default security and integrity levels are valid; otherwise, the user must go through a process similar to the **sl**[8] procedure that exists on the XTS-300.
- Subsequent invocations of the trusted path interrupt application processing. If the user chooses to continue the session, application processing will resume. If the user chooses to logout, the session is terminated.
- The additional software components that compose the Secure LAN Server are transparent to the TCBE and the protocol server.

In order to implement all of these features, we created an application called the Secure LAN Server on the server. It can be divided into two categories of functionality called the Trusted Path Server and the Session Server. The Trusted Path Server (TPS)

---

[8] If a user has the change security level permission, the **sl** command allows the user to change the security and integrity levels of the current session at the trusted path prompt.

behaves like a traditional protocol server and spawns a child process called the Session Server when a new connection is requested. The Session Server has two modes, Authentication and Socket Relay. The Session Server (Authentication) provides an interface between the user's TCBE and the TCB on the remote server to transmit user authentication and session management information. The Session Server (Socket Relay) acts as a secure intermediary between the TCBE client and the protocol server. The interface presented to the client application mimics the interface of the normal protocol server. Similarly, the Session Server (Socket Relay) presents an interface to the protocol server that mimics normal socket behavior.

A protocol server normally binds to a socket that uses a well-known port number. Clients that want to use the protocol server request a connection on the protocol server's listening socket (see Figure 4). The protocol server spawns a child process, which has a different socket, to handle the new connection.

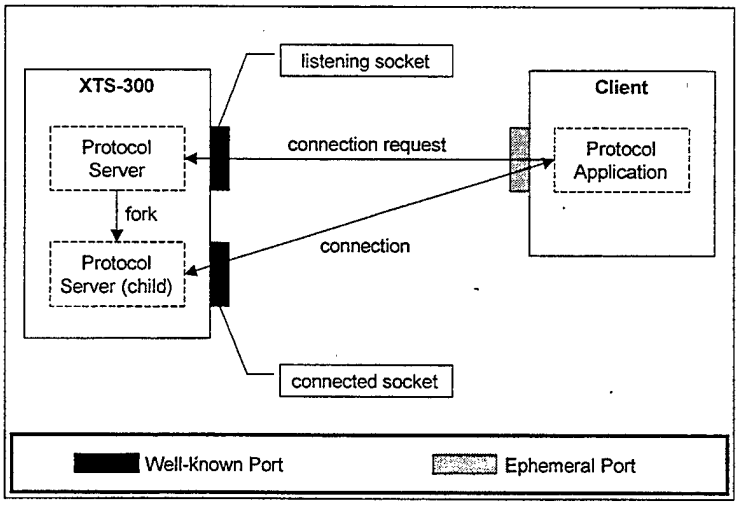

Figure 4. Normal Protocol Server Socket Connection

In our scenario, the TPS binds to a reserved port number and creates a listening socket on which connection requests are received (see Figure 5). By sending a secure attention sequence (SAS) via a trusted computing base extension (TCBE), a user is requesting a trusted path. When a user sends the first SAS, the TPS forks a child process called Session Server. The Session Server creates two types of sockets. The first is a

pseudo-socket to which the protocol server can bind and the second is a real socket to which the TCBE/client application can bind.

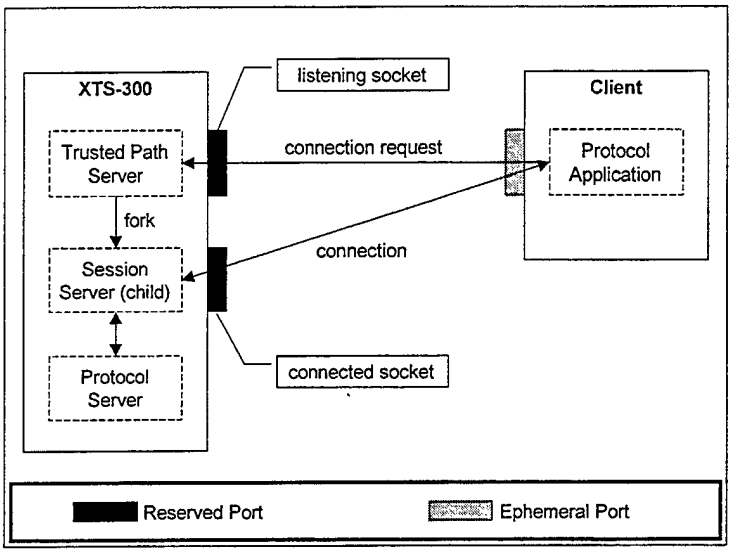

Figure 5. Modified Protocol Server Socket Connection

The Session Server does not establish a trusted path until the TCBE hardware authentication has been completed successfully. Once hardware authentication has been validated, the Session Server (Authentication) creates a trusted path to accept user authentication and session management information. If the user authentication and session level request are valid, the Session Server (Socket Relay) acts as a relay, with optional encryption to protect confidentiality and integrity, between existing protocol servers and the corresponding Trusted Computing Base Extensions. If a user presses the SAK, a SAS is generated and routed to the Session Server (Socket Relay) and a trusted prompt will be displayed at the user's terminal. The trusted prompt will allow the user to logout or continue.

The Secure LAN Server is designed to support request connections from multiple client TCBEs. When different TCBEs try to contact the Trusted Path Server at the fixed connection request location, the TPS creates a separate Session Server to handle each connection request, as shown in Figure 6. Once a connection has been established, any other SASs from a particular TCBE are handled by the Session Server assigned to that

connection. The security and integrity levels of the connection determine with which protocol server the Session Server allows the client to interface.

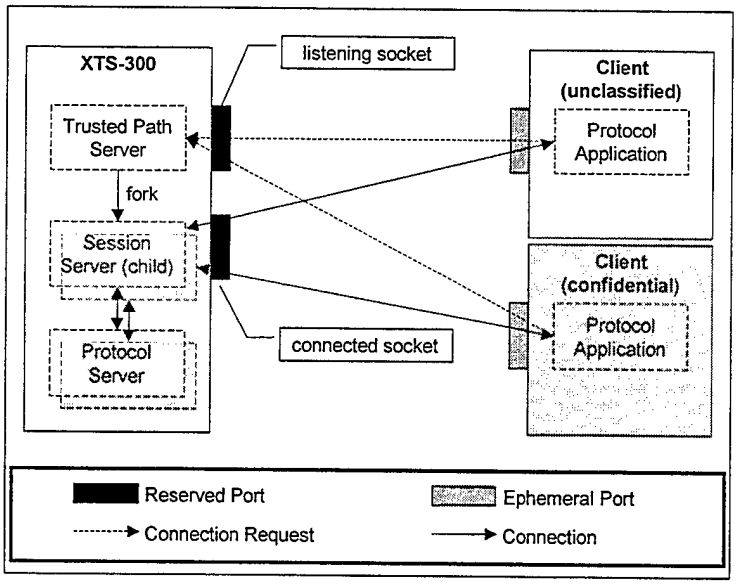

Figure 6. Secure LAN Server with Multiple Clients

Although the goals of the thesis did not change, the design of the software components went through several major revisions. The second section discusses the design process. The final section presents the significant changes in chronological order and the reasons for these changes to illustrate the progression of the design and the steps used to implement that design.

B. DESIGN PROCESS

1. Functional Decomposition

The initial approach in designing the Trusted Path Server (TPS) was functional modularization. We asked questions such as "What occurs next?", "What procedure should be responsible for that functionality?", "How can we communicate via the TCP/IP stack?", "How do we communicate with the protocol server?", which all focused on what functionality is needed at what time. The answers to these questions led to the development of a complex set of control flow diagrams, which are included in this chapter.

The design process began with the TPS, which forks a child process called the Session Server to handle each connection request. The Session Server (Authentication) performs both hardware and user identification and authentication. User identification and authentication is only required to initiate a new session. Hardware identification and authentication is performed by the Session Server every time a secure attention sequence (SAS) is sent from the trusted computing base extension (TCBE). If the session is created successfully, the Session Server (Socket Relay) acts as a relay between the client's TCP connection and the protocol server's pseudo socket connection.

This design approach created modules called Trusted Path Server (TPS), Connection Database (CDB), Session Server (Authentication), Session Server (Socket Relay), and Pseudo Socket. The TPS was to listen and accept socket connections. The CDB maintained a record of the authorized TCBEs, with their associated public key, and whether there was an active session. The Session Server (Authentication) called procedures to perform hardware and user identification and authentication. The Session Server.Socket relay acted as a relay between the client's socket connection and the protocol server's pseudo socket, performing hardware identification and authentication upon receipt of a SAS. The Pseudo Socket emulated the system's socket calls for the protocol server.

2. Object Model Decomposition

After attempting to implement our application using these concepts, it became clear that the multiple interactions between modules were adding to the complexity of the implementation and hence our debugging time. Re-examining the module decomposition using what Parnas [Ref. 19] calls an unconventional approach immediately yielded benefits in understanding the overall project. Our "unconventional" (object oriented) approach was to examine the data stores required by the application, and then provide an interface module for each of the database types thereby minimizing intra-module dependencies and complexity.

This second approach yielded additional software modules called Shared Memory Structure, Shared Memory, Semaphores, and Buffer I/O. The Shared Memory Structure is

responsible for all calls that manipulate data structures in shared memory. The Shared Memory Structure Module is built upon the Shared Memory Module, Buffer I/O, and Semaphores. The Trusted Path Server main procedure calls interfaces to the Shared Memory Structure. The resulting hierarchical structure of dependencies (see Figure 7) yields an application that has proven to be much easier to debug and understand.

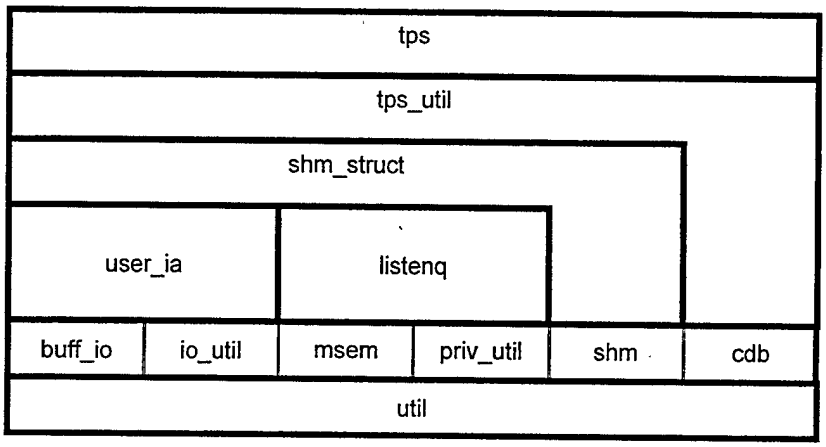

Figure 7. Software Module Dependency Diagram

4. Module Responsibilities

The TPS continues to listen and accept socket connections before forking a child to handle the connection, but the child process acts as a driver for the various data stores. The Session Server (Authentication) passes any socket connection to an identification and authentication procedure that will make requests of the Connection Database and various STOP User Access Databases. If the procedure successfully validates the TCBE and the user, the connection is handled by the Session Server (Socket Relay). The Session Server (Socket Relay) acts as a controlling driver for the various data store modules it must manipulate. The Session Server (Socket Relay) depends on the Shared Memory Structure Module to pass data to the protocol server and on the Buffer I/O Module to pass data to and from the client's socket connection. The Shared Memory Structure also depends on the Buffer I/O module for the to_server and to_client buffers. The reuse of previously debugged code shortened the coding effort by several days. If either data store were to be changed, the Session Server (Socket Relay) would not need significant modification to make the same

function calls and declarations. For example, if FIFO pipes were used in place of shared memory, the Session Server (Socket Relay) could still make calls to a module called Shared Memory Structure. However, the implementation behind the Shared Memory Structure would change considerably (information hiding), albeit the name would be less than descriptive.

The procedural design yielded a framework for understanding the overall goal of the project and for discovering the required databases. The control flow diagrams facilitated real understanding of how a secure attention sequence (SAS) should be handled, and what side effects could be expected when responding to each SAS. The procedural design also produced the first data store module break out (Connection Database) and yielded the modules that act as drivers (TPS, Session Server) during application execution. The object-oriented analysis of the control flow produced an easily understood object-oriented design.

C. DESIGN DECISIONS

During the design process, several discoveries led to the decisions that resulted in the final design that is presented in the next chapter. The discoveries and decisions are presented chronologically in this section to show the progression of the design.

1. Simplification of Secure LAN Server Implementation

For demonstration and development purposes, we designed the Secure LAN Server to function as a stand-alone application, vice relying on *inetd*[9] to start each Session Server. The stand-alone application is called the Trusted Path Server (TPS) and is a socket listener that provides *inetd*-like services for the Session Servers. This decision resulted in two implementation simplifications. We maintained the connection database in memory owned by the TPS, and we were able to execute the Trusted Path Server from the command line.

---

9 Inetd is a daemon associated with UNIX servers using TCP or UDP. It handles most of the startup details of other daemons and is the one process that waits for incoming client requests. [Ref. 20]

Storing the connection database in the memory space of the TPS allowed our implementation to postpone the development of a dynamic database interface, thus saving considerable time in producing the first version for a live demonstration. Since each Session Server is a child of the TPS, the Session Servers each had access to the database pointer, which was required for updates, with no additional coding. Mutual exclusion constructs are not required since each Session Server is responsible for updating only the record associated with the TCBE the Session Server is currently serving. Once fully developed, this database interface will support access by multiple processes, each supporting a different session, while providing mutual exclusion at a record level. We postponed implementing this portion of the connection database during our proof of concept implementation.

Running the Trusted Path Server from the command line speeded the development process by allowing simplified monitoring of the application's behavior via debug statements. Once the application is installed as a trusted daemon, it will not be possible to view run time debug statements and log file functionality will need to be designed and implemented.

2. Amount of Trusted Code

Privileged code implements a privileged subject, which we have defined as a subject that has an integrity level of at least operator. The original design attempted to minimize the amount of privileged code; privileged subjects that are exempt from specific access control rules are called trusted subjects. A discussion of the distinctions between these phrases is presented in Chapter I, Section G.

Consequently, there were several software modules, only one of which was trusted (see Table 1). The TPS daemon, the user identification and authentication module, and the Session Server (active) were originally privileged modules. The process that created the Session Server (active), the Session Creator, was trusted. The Session Creator was the only module that had to communicate with subjects at multiple levels. It was required to take information from a system low module, the user identification and authentication, and create a child process that was possibly at a higher level.

Privileged Subjects	Trusted Subjects
TPS Daemon User Identification and Authentication Session Server (active)	Session Creator

Table 1. Subjects in Original Design

While porting our first network application to the XTS-300, we discovered that a network application must be at the same level as the TCP/IP daemon (system low) or be trusted in order to utilize sockets. Considering this new requirement, alternative solutions had to be considered. The first, instantiating new TCP/IP stacks for each level, was quickly eliminated since it would be excessively resource intensive.

The next solution we examined was designing the TPS as a privileged process that creates a trusted child process, the Session Server. In order to be able to receive information from the TCP/IP connection and create a child process to support a session that was at any level higher than system low, the TPS was required to be a privileged process. As a child process, the Session Server inherits the characteristics of the parent and is able to handle all further communication with the TCBE/client application pair.

In order to communicate between the TCP/IP stack and any protocol servers operating above security level zero and integrity level three (the level of the TCP/IP stack), the module previously called Session Server (active) would have to be a trusted subject. Separating it from the Session Creator would no longer minimize the number of trusted subjects; consequently, the two modules were put into one process and the resulting trusted subject was called the Session Server. Table 2 depicts the results of the design change.

Privileged Subjects	Trusted Subjects
TPS	Session Server

Table 2. Subjects in Final Design

3. Secure Attention Sequence (SAS) Handling

a. Alleviating Denial of Service Attacks #1

Initially, the TPS was designed to receive each SAS destined for the system and perform the TCBE hardware identification and authentication (HW_IA) before it determined the correct process to forward the SAS signal to for action. The initial intent was to provide consistent handling of the SAS, but the overhead associated with verifying the TCBE hardware ID for each SAS represented a potential choke point. Since each active session had a direct connection between the Session Server and its respective TCBE, we decided to have the SASs sent directly to the Session Server currently acting as the TCBE's controlling active process. This decision eliminated the overhead that was introduced by having the TPS determine which Session Server should handle each SAS.

In addition to adversely impacting system performance, the choke point also introduced a vulnerability to denial of service attacks. In order to alleviate the problem, the TPS was redesigned to spawn a child process to handle hardware and user authentication procedures every time it received a SAS. The Session Server performs the TCBE hardware ID before further processing the SAS.

b. Preventing Multiple Session Servers for a Single TCBE

Since every SAS received by the TPS was now generating a Session Server, checks against the connection database were introduced into the system to ensure that the SASs associated with an active session were forwarded to the right child process. The TCBE hardware ID procedure serves two purposes. If the hardware ID is not valid, it returns an INVALID_ID; if the hardware ID is valid, it returns the controlling active process ID (CAPID) and saves the hardware ID in the parameter that was passed by reference. If the CAPID is not equal to TPS_CONTROL, there is an active session and the Session Server (Authentication) passes the SAS to the CAPID and calls End Session.

To maintain the integrity of the connection database data, a critical region is defined in the code that provides functions to manipulate the connection database. The

critical region begins with the TCBE hardware ID procedure and ends after the conditional structure that updates the CDB if the result of the TCBE hardware ID procedure is TPS_CONTROL. We determined that the critical region was required to prevent multiple processes from one TCBE from receiving a TPS_CONTROL and attempting to update the Connection Database (CDB). Now only one process will receive TPS_CONTROL and update the CDB with its process ID as the CAPID for the TCBE that sent the SAS. The other processes that were created because of SASs from the same TCBE will receive a SESSION_ACTIVE and forward their SAS to the process listed as the CAPID.

c. Alleviating Denial of Service Attacks #2

Creating a Session Server every time a SAS is received presented another problem; the system's vulnerability to a denial of service attack increased. Reviewing the flow of information that was occurring revealed a possible solution: the TPS only needed to handle SASs for connection requests. SASs for established connections could be routed directly to the Session Server responsible for the session.

Although it is still possible that a flood of connection requests could constitute a valid Denial of Service attack, a properly configured LAN can significantly reduce the attack's effectiveness. IP filtering at the incoming router, and the resulting restricted IP addresses, limit the number of users who are capable of performing this type of Denial of Service attack and make identifying the culprits much easier.[10]

d. Handling Multiplexing Issues

Because of the possible delay in creating a Session Server when the first SAS is handled, there could be other SASs from the same TCBE that are routed to the TPS queue before the connection is actually established. In this situation, the approach is very similar to that presented in the previous section on "Preventing Multiple Session Servers for

---

10 Denial of service attacks are still possible. However, we have proposed a mechanism by which we can limit the number of possible attackers. If the LAN is configured as intended, the router providing external communications should be on the far side of the high assurance server from the LAN clients.

a Single TCBE". However, only the SASs that are received in the time period between the first SAS being received and the connection being established end up in the TPS queue. Any SASs received after the connection is established are forwarded directly to the Session Server responsible for the connection. Therefore, the possibility discussed in the section on "Alleviating Denial of Service Attacks" is diminished.

III. FINAL DESIGN

This section presents details of the final design by presenting the functional diagrams for each portion of the Trusted Path Server (TPS) and the Session Server. Figure 8 illustrates a high level diagram of the Secure LAN Server components. Figure 9 shows the high-level transition control diagram of the required components. The Connection Database is a pivotal component that does not appear in Figures 8 or 9 because it is part of the TPS initialization. The characteristics of each component will be explained in its corresponding section.

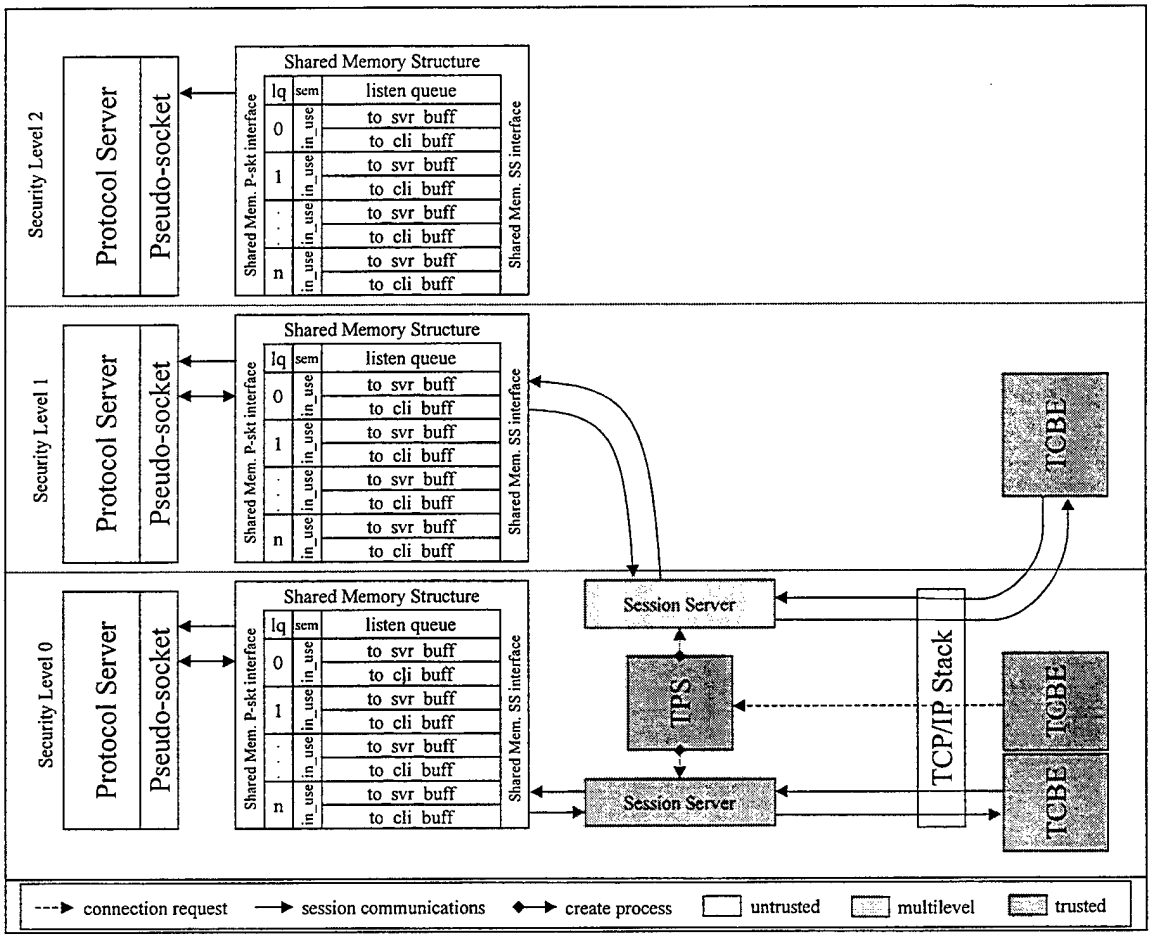

Figure 8. Secure LAN Server Overview

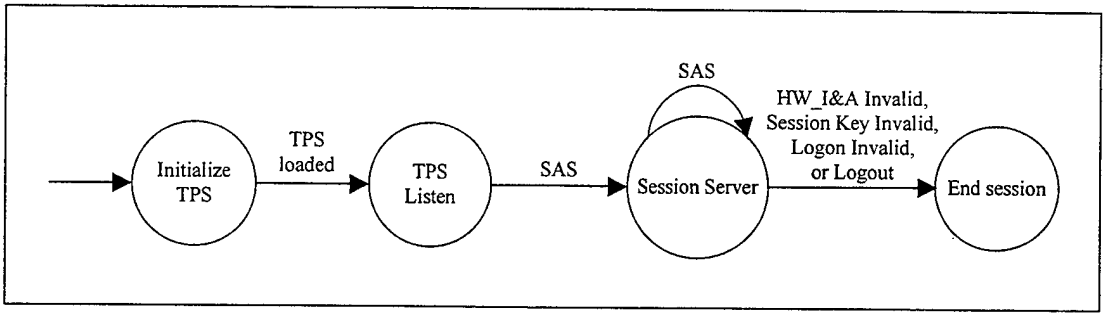

Figure 9. Transition Control Diagram

A. COMPONENTS

1. Connection Database (CDB)

The CDB is vital to the implementation of the TPS and Session Server. It is initialized from a protected file when the TPS is started, and maintained in memory as long as the TPS exists. The code associated with maintaining the CDB must be trusted to prevent unauthorized modifications.

For demonstration purposes, the CDB was designed to allow the system administrator to make additions, deletions, and modifications to the CDB records by directly modifying the initialization file. This means that changes only take effect when the TPS restarts. Future options are discussed in Chapter IV, Section B.

The structure of the CDB is displayed in Figure 10. If the TCBE hardware ID is valid, the CAPID field holds the process ID (PID) of the process that is responsible for the current session. If the controlling active process ID (CAPID) is TPS_CONTROL, there is no active session for the TCBE.

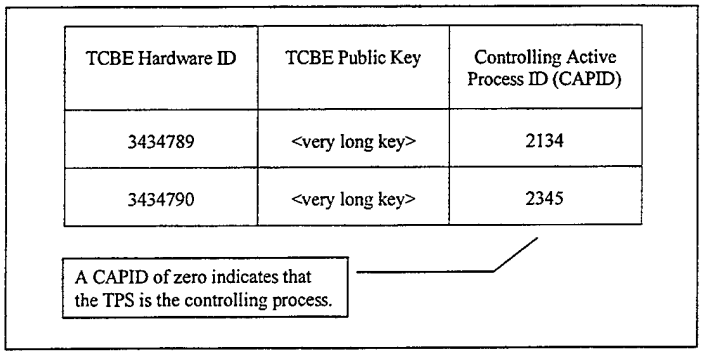

Figure 10. Connection Database Structure

2. Secure Attention Sequence (SAS)

The secure attention sequence (SAS) (see Figure 11) is created by the TCBE in response to a user SAK. The SAS is used to initiate trusted path negotiation. The TCBE is not required to monitor the data stream from the client PC for the reserved sequence of keystrokes since they can only be entered under control of the TCBE. This is true because a valid SAS is signed using the TCBE's private key by the TCBE prior to transmission to the Secure LAN Server. It is unlikely that the untrusted software on the PC client will be capable of spoofing the SAS without access to the private key of the TCBE. The likelihood of this event will be determined when an appropriate public-key algorithm is selected.

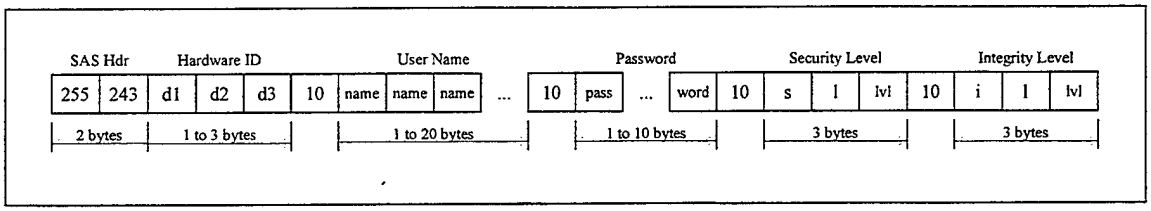

Figure 11. Secure Attention Sequence

When a user enters a reserved sequence of keystrokes, the TCBE stops the data flow from the client PC and sends a SAS to the XTS-300. The Secure LAN Server must monitor all inbound packets for the SAS header sequence and react accordingly when a SAS header is detected. The reaction depends upon whether the SAS is part of the initial session request or occurs during an active session. These reactions are covered in more depth in the TPS and Session Server sections.

Requiring the Secure LAN Server to monitor the individual bytes should add negligible overhead while simplifying the design of the Session Server. Minimal extra coding is required to add a conditional branch if the SAS header is detected. If TCP out-of-band data or some other mechanism of signaling the Session Server were to be used, the complexity of the Session Server and the TCBE will be increased. Both of these components have design goals of limiting the amount of code required. The reason for minimizing code in the Session Server and TCBE is to simplify the evaluation in accordance with the TNI [Ref. 7], and to minimize the final hardware implementation of the TCBE. Additionally, monitoring of the inbound data may be easier if a block cipher [Ref. 9: p. 57] is selected and the TCBE is required to send a SAS on a block boundary. The selection of a symmetric cryptographic algorithm is further discussed in Chapter IV under future research.

Once the trusted path is established, only trusted path communication is allowed. Data flow from the client PC does not resume until the trusted path communication is over and the TCBE resumes transmitting data from the client PC. This is discussed further in the Trusted Prompt section of the Final Design chapter.

The SAS contains the TCBE hardware identification (ID) and a nonce[11] that is encrypted with the Secure LAN Server public-key that are used to establish a trusted path. When the SAS arrives at the XTS-300, either the TPS or the Session Server that is supporting the current session handles it; the TPS and Session Server sections discuss these procedures in more depth.

3. Trusted Path Server (TPS)

The TPS is a program that is started from the trusted prompt and runs in the Operating System Services (OSS) domain of the XTS-300. Although not currently a daemon, we are simulating one in the TPS. Execution of the TPS begins with an

---

11 A nonce is a unique character string often representing time used in cryptography to provide protection against replay attacks. [Ref. 10]

initialization process that consists of loading the Connection Database into memory from a protected file and binding the TPS process to the port that is reserved for the Trusted Path Server. When these steps have been successfully completed, the TPS enters a listening state (see Figure 12) in which it blocks until a user initiates a connection by sending a secure attention sequence (SAS) via the TCBE. As connection requests are received, they are placed in a first-in-first-out (FIFO) queue maintained by the TCP/IP stack. The size of the pending request queue is a design parameter (we chose 5), and can be changed in the TPS source file. Future work could place this in a configuration setup file accessible the system administrator.

Once a connection is established, the TPS creates a child process called the Session Server to handle all further communications for that connection, forwards the SAS, and continues to monitor for any other pending connection requests. If there are requests present in the queue, the TPS handles them in a first-in-first-out (FIFO) manner as discussed above; if there are no other requests pending, the TPS re-enters its listening state.

Designing the TPS as a command line program shortened the modification and testing cycle. Extending the design for the TPS to function as a daemon is discussed in Chapter IV under future research.

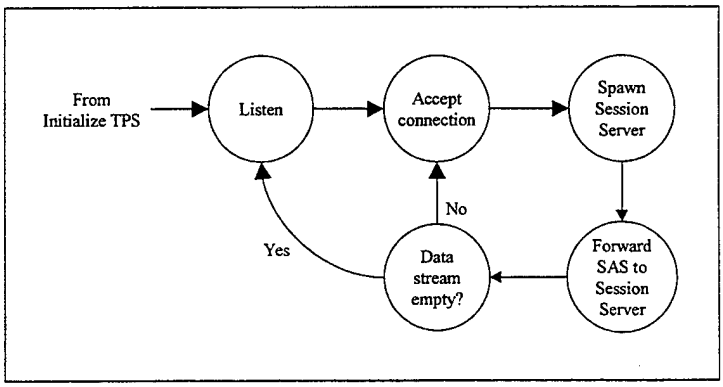

Figure 12. Trusted Path Server Listen Mode

4. Session Server

a. Session Server (Authentication)

The first thing the Session Server (Authentication) (see Figure 13) does is to determine whether the hardware ID of the TCBE that made the connection request is valid. This is accomplished during the Hardware Identification and Authentication (HW I&A) that is discussed in more detail in a later section. If the hardware ID is invalid, the Session Server (Authentication) calls End Session. There are two possible branches if the hardware ID is valid; one occurs if there is an active session, and the other occurs if there is no active session. If there is no active session for the TCBE, the value in the CAPID field is TPS_CONTROL. In this case, the Session Server (Authentication) updates the controlling active process ID (CAPID) of the TCBE in the connection database (CDB) with its process ID. Once this update occurs, an active session has been established and the second branch occurs. Any subsequent SAS from the same TCBE is handled directly by the Session Server.

Each SAS causes the HW_IA procedure to be invoked; any time a HW_IA fails, the connection is terminated. If a SAS is received by the Session Server (Authentication), it exits the current procedure and repeats hardware identification and authentication. For example, if the Session Server (Authentication) has begun, but not completed, user identification and authentication, a SAS will cause the Session Server (Authentication) to discontinue "User I&A" and being again with the bubble labeled "HW_I&A initial session." SAS transitions are shown only where they are allowed.

The design includes an option for negotiating a one-time session key to encrypt ensuing session communications. If the encryption option is selected, a session key is negotiated and used; the details of the negotiation are covered in more detail in the section on Negotiate Session Key. Regardless of whether the encryption option is selected, the Session Server (Authentication) begins the user login procedure, which is discussed in more detail in its own section. If the user identification and authentication fails, the Session

Server (Authentication) calls End Session; if it succeeds, the process changes modes from Session Server (Authentication) to Session Server (Socket Relay), effectively beginning the user's active session.

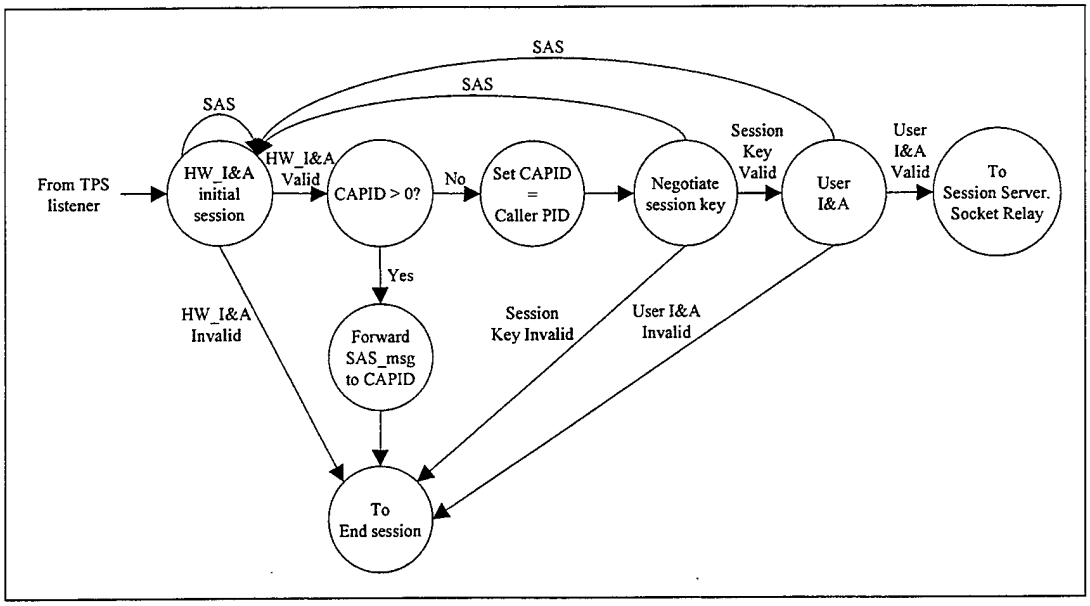

Figure 13. Session Server (Authentication)

b. Session Server (Socket Relay)

Separate Session Servers (Socket Relay) (see Figure 14) are responsible for establishing and maintaining the secure sessions for each TCBE and protocol. Once the Session Server (Socket Relay) has control of the session, it blocks while waiting for data from the client PC. Once it receives data, the Session Server (Socket Relay) calls Session Relay, which is discussed in more detail in its own section. The Session Relay is active as long as data is being received; when the data stream is empty, the Session Relay returns control to the Session Server (Socket Relay).

When the Session Server (Socket Relay) receives a SAS, a Hardware Identification and Authentication (HW I&A) occurs to ensure that the TCBE is still valid. If the HW I&A fails, the Session Server (Socket Relay) calls End Session. If the HW I&A is successful, the user will see a Trusted Prompt at his PC. The functionality of the Trusted Prompt is

discussed in more detail in a later section. If the user chooses to continue his current session, the Session Server is restored to its previous state. If the user chooses to logout, the Session Server (Socket Relay) calls End Session.

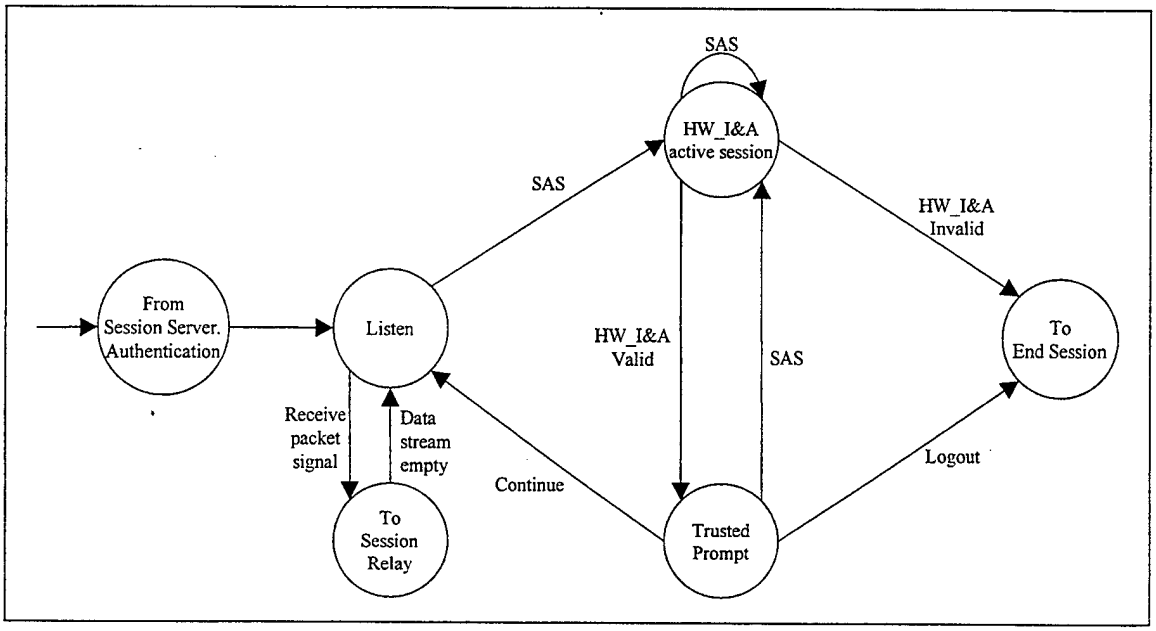

Figure 14. Session Server (Socket Relay)

5. End Session

End Session (see Figure 15) is a procedure that cleans up when a connection is terminated. The CAPID value in the CDB for the TCBE is reset to TPS_CONTROL (indicating that there is no active session) and the Session Server process for the connection exits with the appropriate status code.

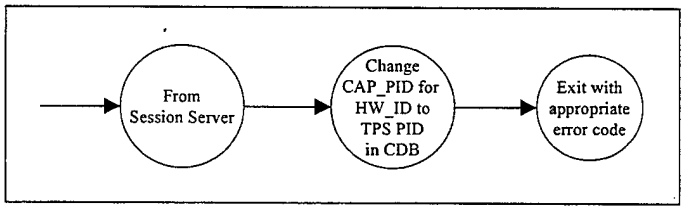

Figure 15. End Session

6. Hardware Identification and Authentication (HW I&A)

TCBE Hardware Identification and Authentication (see Figure 16) is a crucial component of the Trusted Path. Without verification of the identity of the hardware, there is no guarantee that the client identity is not being spoofed. The HW_IA is called from both the Session Server (Authentication) and the Session Server (Socket Relay). During this development phase, the TCBE Hardware Identification and Authentication encryption was simulated. The HW_IA consisted of verifying the TCBE hardware identification number against those present in the Connection Database (CDB). If there was a record with a corresponding TCBE hardware identification number, the TCBE was considered authenticated and the controlling active process identification (CAPID) was returned. Implementation of hardware authentication was reserved for future research efforts as discussed in Chapter IV, Section B.

Both public-key and symmetric-key cryptography could be used to encrypt the communications between the TCBE and the Secure LAN Server, but there are several reasons that we chose public-key cryptography. Symmetric-key algorithms are simpler and faster, but the key must be exchanged in a secure out-of-band manner. Alternatively, public-key encryption allows the public key to be distributed in a non-secure way and the private key is never transmitted.

Public-key cryptography can be used to simultaneously protect the secrecy of the TCBE hardware ID included in the SAS and provide authentication between the TCBE and the Secure LAN Server. The TCBE signs the SAS using its private key and then uses the Secure LAN Server's public-key to encrypt the SAS before it sends it over the network. HW_IA decrypts the SAS using the Secure LAN Server's private key and looks in the CDB to see if there is a matching TCBE hardware ID. If there is not, the HW_IA returns an INVALID_ID. If there is, the HW_IA returns the CAPID associated with the TCBE hardware ID and copies the hardware ID into the hardware ID parameter passed by reference.

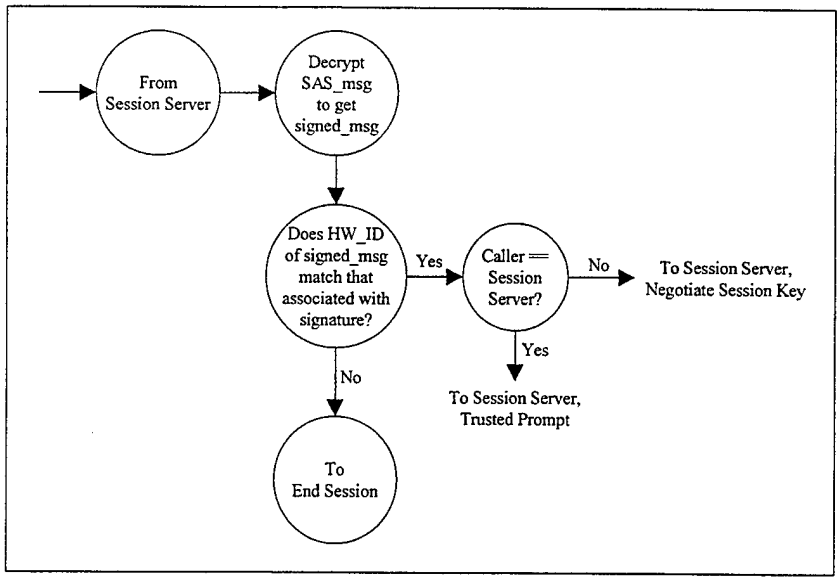

Figure 16. Hardware Identification and Authentication

7. Negotiate Session Key

The option to negotiate a session key (see Figure 17) has been included in the design, although it has not been implemented as a part of this thesis. In environments that do not provide protection against malicious eavesdropping, a one-time session key provides confidentiality for the communications between the client and the server. A symmetric-key encryption algorithm was chosen over public-key encryption because it is faster and thus more appropriate for bulk data encryption. A public-key exchange algorithm, Oakley, was chosen as the method for calculating the one-time session key for symmetric encryption because it does not require the session key to be transmitted over the network.

A test message is sent as an automatic communications check to ensure that both the TCBE and the Session Server correctly calculated the one-time session key. The test message should be long enough to accurately exercise the diffusion property of the encryption algorithm. If the Session Server is unable to decrypt the test message and retrieve the expected plain text, then the connection is terminated. If the TPS can properly decrypt the test message, the user begins identification and authentication procedures.

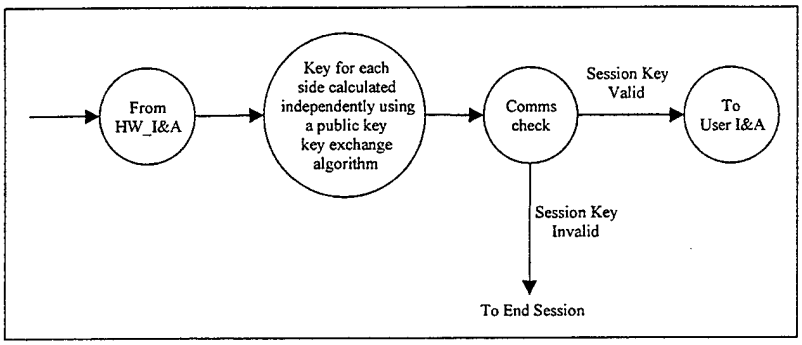

Figure 17. Negotiate Session Key

8. User Identification and Authentication

The user identification and authentication procedure accepts user name, password, and session level information from the user via the trusted path. The current implementation for verifying the user information against the User Access Databases associated with the STOP operating system is stubbed out. Future implementations are discussed in Chapter IV, Section B.

9. Trusted Prompt

The Trusted Prompt is differentiated from the login prompt. While both prompts use the trusted path as the communications conduit, the login prompt occurs once, immediately after the connection has been established. The Trusted Prompt can occur at any time, after both the connection and the session have been established. The Trusted Prompt (see Figure 18) developed as part of this thesis attempts to emulate the appearance of the trusted prompt that is normally associated with the XTS-300. Once a session has been established and the user is successfully authenticated, a SAS from the TCBE will initiate a trusted prompt. A session message that contains the session security and integrity levels is assembled to send to the client PC. As all trusted path communications are encrypted, the session message must be encrypted. When the session message reaches the TCBE, it is decrypted and the contents are displayed under control of the TCBE.

At this point, the trusted path has been established and the Trusted Prompt on the remote server is waiting to receive a command from the user via the TCBE. In order to

demonstrate the validity of our approach, our trusted prompt supports a subset of the usual trusted path functions; the user may continue or logout. The command *continue* reattaches the user's session in its previous state; the command *logout* terminates the session. Any other entry will cause the user to be prompted for a command input again. The restrictions placed on the commands permitted over the Ethernet trusted path prevent administrative functions from being performed from client PCs. This feature reduces the possibility that the system could be subverted by a user from an external location.

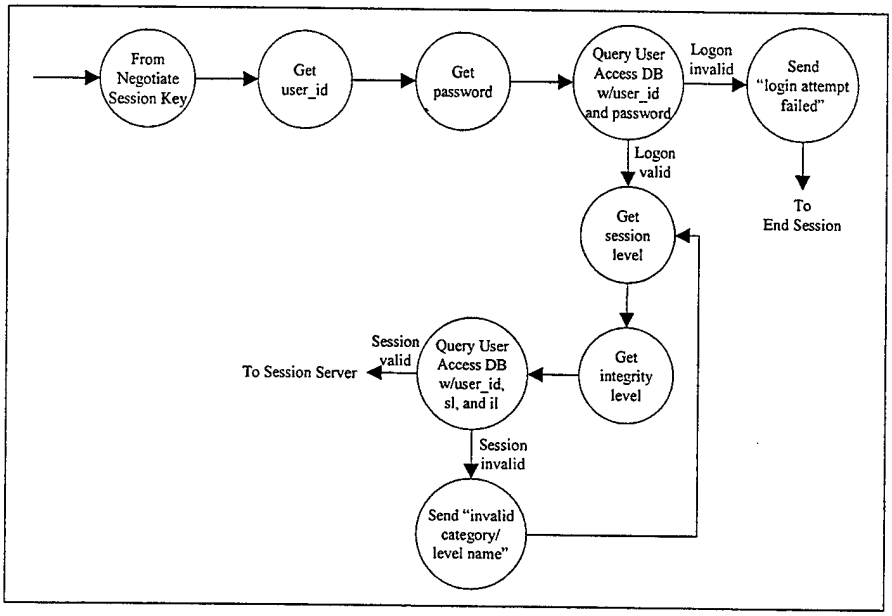

Figure 18. Trusted Prompt

10. Session Relay

The Session Relay (see Figure 19) is called by the Session Server (Socket Relay) when there are incoming packets on a connection. When the data packet reaches the Session Relay, the TCP/IP header has been stripped away, but the packet is still encrypted. The Session Relay is responsible for decrypting the packet and forwarding it to the appropriate protocol server via a pseudo-socket interface.[12] If, for some reason, the appropriate protocol

12 The details of the pseudo-socket interface and the shared memory structure are discussed in the implementation chapter of this thesis. However, it is worthwhile to note that there is only one shared memory structure per

server has not been started, the Session Relay is responsible for starting and opening a communications path with it. The Session Relay checks the data stream for more input to handle; if the data stream is empty, it returns control to the Session Server (Socket Relay).

For demonstration purposes, our thesis implements a Session Relay that supports one protocol, but the design can be extended to multiplex between multiple protocols. Future developments are discussed in Chapter IV, Section B.

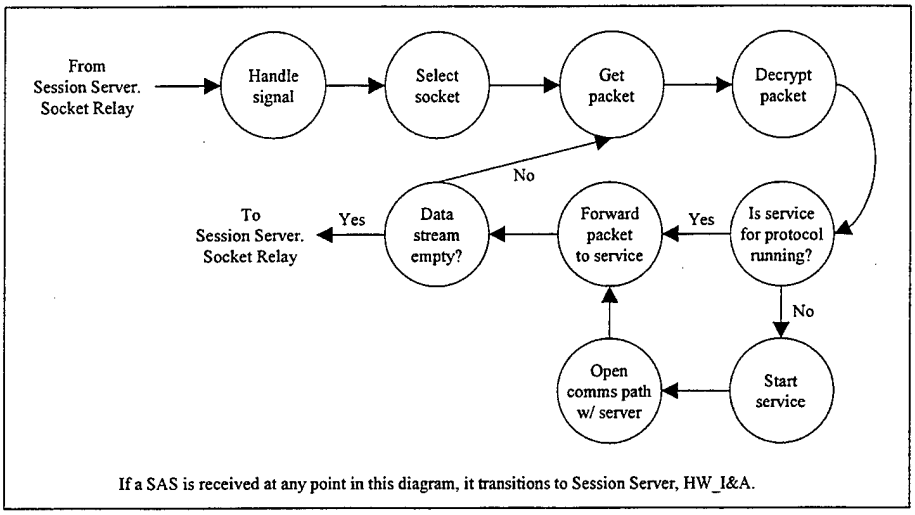

Figure 19. Session Relay

11. Audit

The act of recording events in the audit trail is not noted on any of the previous diagrams to prevent unnecessary distraction or confusion. The events that are audited can be selected by the system administrator, since our system is based on the XTS-300 which allows the auditing of all or selected events. The audit records that our system might generate are shown in Table 3 [Ref. 18: pp. 21-26].

---

sensitivity level. This shared memory structure is used to allow communication between a secure session server and the protocol servers associated with a sensitive level.

No.	Audit Event
8	device open
11	device close
132	device start error
68	object close
70	object creation
72	object deletion
74	object open
13	IPC message sent
17	process creation
19	process deletion
20	process fork
21	process owner change
22	process privilege change
32	shared segment map
33	shared segment unmap
34	semaphore object
77	socket open close
78	socket bind failure
79	socket connect
80	socket accept
83	Internet inbound connect failure
131	**cup** (change user password) command
134	login
135	logout
136	operator command
140	**sl** (change session level) command

Table 3. Audit Events

B. IMPLEMENTATION PHASES

1. Background Information

The XTS-300 has four primary software components: the Security Kernel, TCB System Services (TSS), Trusted Software, and the Commodity Application System Software (CASS). [Ref. 21] The Security Kernel provides basic system operating services and enforces system security policy. The TSS software provides general trusted services to XTS-300 application and system software. Trusted Software provides additional security services outside the Security Kernel. CASS provides an environment on the XTS-300 for the execution of UNIX-based application programs.

A ring mechanism is also provided to augment the security of the XTS-300 system (see Figure 20). It is used to isolate portions of a process from being tampered with. Ring 0 is reserved for the Security Kernel and is the most privileged ring. Ring 1 is reserved for the TSS. Ring 2 is reserved for Trusted Software, CASS, and site-developed trusted processes, and is less privileged. Ring 3 is reserved for user processes and is the least privileged. [Ref. 21]

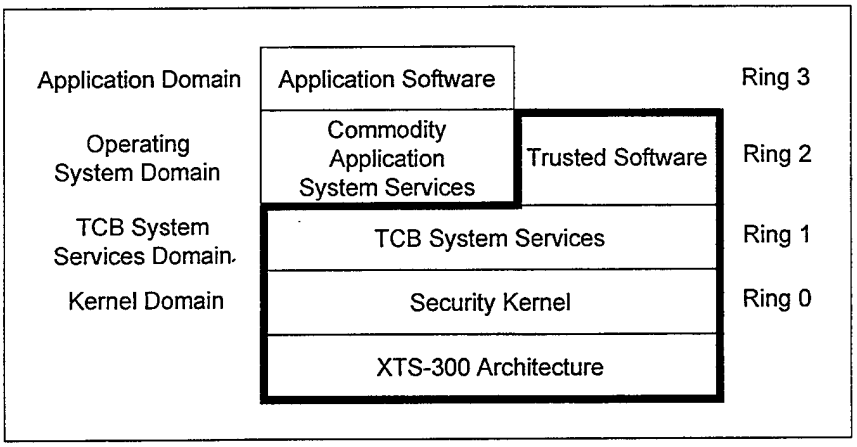

Figure 20. XTS-300 System Diagram

"The policy that the XTS-300 enforces is the DoD policy on multi-level secure computing as formalized in the National Computer Security Center (NCSC) approved Bell-LaPadula mathematical model." [Ref. 21] The trusted computing base (TCB) enforces the

mandatory rules: the simple security property, *-property for secrecy, the simple integrity property, and the *-property for integrity. The enforcement of these rules is based on comparison between the clearance of the user and the labels associated with the objects in the system. Clearance and labels are composed of security levels (**sl**) and integrity levels (**il**). A security level is the combination of a security classification and a set of security compartments. An integrity level is the combination of an integrity classification and a set of integrity compartments. Security level 0 (sl0) means the lowest classification level. Similarly, integrity level 0 (il0) means the lowest integrity level.

The XTS-300 is designed to support most of the UNIX System V Release 3 system calls in Ring 3 and a significant subset of the System V Release 3 system calls in Ring 2. However, the differences were great enough that we experienced a very steep learning curve during the implementation of the Trusted Path Server in Ring 2 of the system. We decided to approach the problem by decomposing it into small steps; each of which provided incremental progress toward the final goal. The phases are outlined here, although not in comprehensive detail.

2. Porting an Echo Server to the XTS-300

The first step was choosing an initial program to port to the XTS-300 and successfully doing so. The components of our final program provided the basic functionality of accepting information from a pseudo-TCBE and sending information back to the TCBE/client PC. An echo server provides a similar service, and it was chosen as the program to port.

We used code from Stevens [Ref. 20: pp. 112-115] as the basis of our echo server and client. Since the author defined some wrapper functions for the various socket system calls in the name of portability, we had to replace them with the corresponding system calls that were supported on the XTS-300. Some of the functions used in the code are not supported by the XTS-300, and had to be replaced with equivalent functions that are supported. Another problem was that Stevens had consolidated all of the *#includes* into one huge file called "unp.h". Mapping Stevens' *#includes* to the sometimes non-standard XTS-

300 includes required careful attention to the XTS-300 documentation and Steven's documentation. At this point, the code compiled successfully and ran in the application layer (Ring 3). While the echo server was active, a client could telnet to the port associated with the echo server, enter a string and have it echoed back to his terminal.

3. Accepting Manually Entered SAS from Pseudo-TCBE

At this point, we had a working echo server between the TCBE and the TPS and wanted to extend the functionality of the TPS to accept a manually entered secure attention sequence from the TCBE before it spawned a child process. First, we had to define what the secure attention sequence (SAS) from the TCBE was going to be. Since the existing telnet option for the XTS-300 requires a telnet break sequence to initiate a secure attention key (SAK), we decided to require the same sequence.

As we tried to determine the composition of a telnet break sequence, we ran into some inconsistencies. Although the break sequence itself is defined as an IAC[13] followed by a BRK (255 243) in RFC 854, some telnet programs insert a line feed after the BRK. To maintain consistency, we created a pseudo trusted computing base extension (TCBE) that consistently sent the desired sequence of bytes to the Trusted Path Server (TPS). When the pseudo-TCBE ran, it sent the sequence 255 243 digit [digit] [digit] 10 (IAC BRK digit [digit] [digit] LINEFEED), where digit [digit] [digit] represents the hardware identification number with one to three digits.

The TPS code parsed the message upon receipt. If the first two ASCII representations were not 255 and 243, the secure attention sequence (SAS) would fail and the Session Server would terminate. The call to the procedure that checks the validity of the SAS against the Connection Database (CDB) was stubbed out and hard-coded to return successfully each time. The program was successfully establishing connections and parsing the SAS when it was received at this stage of the development.

---

13 Interpret as command (IAC) is an escape character for telnet that is always followed by a command byte.

4. Creating the Connection Database

In order to be able to expand the stubbed out procedure that was verifying the identification number of the trusted computing base extension (TCBE) against the Connection Database (CDB), we had to create the CDB. We started by defining the format of the initialization file. Each line of the file would contain a record associated with a specific TCBE. Each record in the initialization file contains two elements, the TCBE's hardware identification number (one byte) and the TCBE's public key (the length is PKI dependent), separated by a comma and terminated by a carriage return. Since the number of TCBE clients for the demonstration is expected to be relatively small, we decided to maintain the entire database in memory and initialize it at TPS startup. During execution, each record contains an additional field to maintain the controlling active process identification (CAPID). This information is only pertinent at run time and is always initialized to TPS_CONTROL to reflect the TPS as the controlling active process.

Keeping the connection database in memory will limit disk accesses and improve TPS response time. The ultimate impact on system memory resources will depend largely on the size of the public key used to uniquely identify each TCBE and the number of TCBE clients to be served. For example, if there were 100 entries in the CDB and public keys were each 1000 bits in length, then the CDB would consume 12,600 bytes of memory. Determining the public-key infrastructure has been left as future work.

5. Checking Hardware Identification from Pseudo-TCBE

Once the Connection Database (CDB) was initialized and in memory, we could develop the stubbed out procedure that was responsible for checking the validity of the TCBE hardware identification. The hardware identification check is responsible for two things: verifying the validity of the TCBE hardware identification number and returning the controlling active process identification (CAPID) if the hardware identification number is valid. If the hardware identification is not valid, the procedure returns an INVALID_ID.

The hardware identification check is performed as part of the secure attention sequence (SAS) processing. If the hardware identification check is not valid, the SAS is

rejected and the connection is terminated. Now the program is capable of selectively supporting connections based on the validity of the SAS, which must include a valid TCBE hardware identification number.

6. Creating a Loop-back in the Session Server to the Pseudo-TCBE

To maintain our controlled development, we decided to take an intermediate step between the current program and the next stage, which was introducing the full relay capability to the Session Server (Socket Relay). The full relay would allow the Session Server (Socket Relay) to receive information from the TCBE, forward that information to the protocol server, receive information from the protocol server and return the protocol server's response to the TCBE. The intermediate step was to ignore the protocol server and simply have the Session Server (Socket Relay) forward all data received back to the TCBE. In this manner, the pseudo-TCBE would see echo server functionality while we tested the first half of the data flow. The loop-back would prove that the Session Server.Socket relay was able to receive and send input from and to the client application over a TCP/IP socket, the pseudo-TCBE in this case.

We were able to reuse much of the code from the original echo server that we ported to the XTS-300 for this phase. One procedure in the code was called str_echo and its purpose was to read data from a socket and then write it back to the same socket. By inserting a call to this procedure after the Session Server completed the hardware authentication, a loop-back was created in the Session Server (Socket Relay). The program can selectively support connections based on the validity of the SAS and provide an echo server function to the pseudo-TCBE.

7. Providing Interface between Echo Server and Session Server

This stage proved to be one of the most challenging. We had to create a pseudo-socket library for the echo server that would provide socket-like functionality for sockets that were simulated by some other construct. One of the first design decisions that had to be made was to decide what would be used to simulate the socket connection between the Session Server (Socket Relay) and the echo server. Since the pseudo-socket had to be able

to communicate between an OSS domain program (the Session Server (Socket Relay)) and an Applications domain program (the echo server), an Inter-Process Communication (IPC) mechanism was needed. The XTS-300 placed limitations on which IPC mechanisms could be used. The eligible IPC mechanisms that were appropriate for bulk data transfer between processes were FIFOs, shared files, and shared memory. Shared files were ruled out immediately because of the latency file I/O would impose.

Our initial intent was to implement a pseudo-socket library that did not require any modifications to the echo server, so we looked at the remaining IPC mechanisms with regards to their implementation. Although the FIFOs were very socket-like (i.e. a socket descriptor is provided), communication between the Session Server (Socket Relay) and the echo server would require two FIFOs to simulate one socket connection, one for data flow in each direction. The result of this requirement dictated the use of two file descriptors when the pseudo-socket was created, instead of the one that would be expected from a normal socket connection. Although the two file descriptors could have been virtualized into one pseudo-socket, it would have required modifying the echo server to call read and write functions specific to the virtual pseudo-socket, which was contrary to our initial objective.

Since the mechanisms for using shared memory were even more divergent from those for using sockets, we knew that we could not use shared memory without modifying the echo server. Therefore, our goal of no modification changed to one of minimal modification of the echo server. At this point, we began looking at the benefits and drawbacks of the FIFOs and shared memory before deciding which one to implement.

FIFOs are relatively easy to implement, but are less efficient than shared memory. Implementing FIFOs to simulate sockets would have required six copies of the data (see Figure 21) for each round trip from the client XTS-300's TCP/IP stack through the echo server and back to the XTS-300's TCP/IP stack. Conversely, shared memory is more complex to implement, but only requires between two and four copies (see Figure 22). In the interest of having a more efficient program, we decided to use shared memory to simulate sockets.

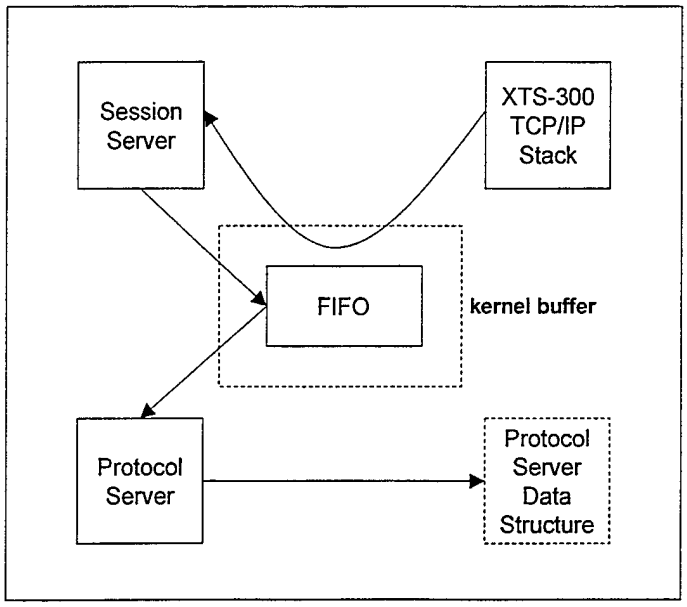

Figure 21. FIFO Data Flow

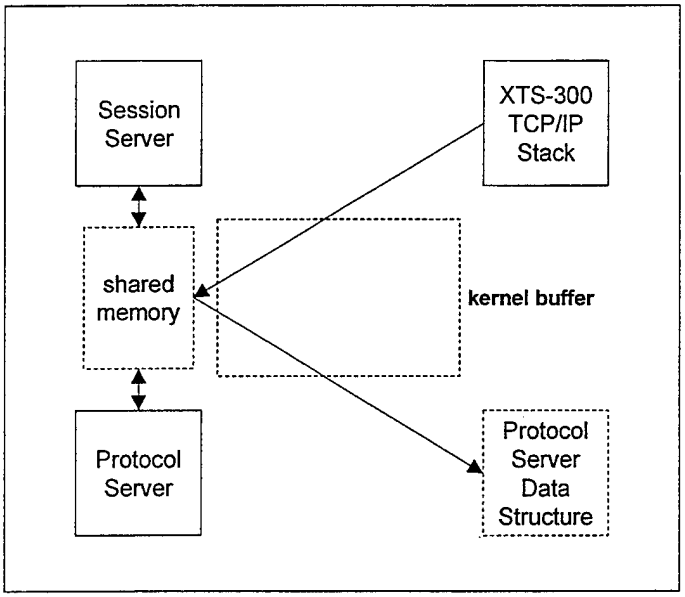

Figure 22. Shared Memory Data Flow [Ref. 22]

The modifications that had to be made to the echo server as the result of our design decision were minimal. The **read, write,** and **close** calls associated with sockets were changed to **my_read, my_write,** and **my_close** calls because we could not overload system

calls in *libc*[14] without a significant operating system redesign. The procedure calls defined in the STOP operating system's socket.h file remained the same. They were replicated, with different internal functions, to provide pseudo-socket functionality to the protocol server via our pseudo-socket file that is used in place of the normal files.

The next step in providing the interface between the echo server and the Session Server (Socket Relay) was to implement shared memory. In order to prevent collisions, we used semaphores to provide mutual exclusion for all of the reads and writes to a particular pseudo-socket connection in shared memory. The constraints imposed by the XTS-300 required that the mandatory access control (MAC) levels of the two processes sharing memory be identical or that one of the processes be trusted and be granted privilege to transcend the MAC enforced by the STOP operating system. Consequently, the Session Server (Socket Relay) is a multilevel process that provides the required MAC exemptions when reading to or writing from a shared memory segment at any level other than security level zero (sl0) and integrity level three (il3).

The procedure calls that occur when a protocol server is preparing to accept connections with normal sockets are:

- socket – if successful, returns a socket descriptor
- bind – if successful, assigns a local protocol address to a socket
- listen – if successful, converts an unconnected socket into a passive socket; indicates that the kernel should accept incoming connection requests directed to this socket
- accept – if successful, returns the new socket descriptor created by the kernel for the connected socket; if the connection queue is empty, the calling process is put to sleep

The corresponding calls that occur when the protocol server is going to accept connections using pseudo-sockets are:

---

[14] libc is the standard UNIX system library.

- socket – creates shared memory segment at MAC level of the calling process; returns the listen queue socket descriptor
    - The key that is a required parameter to the socket call for a process to create or open a shared memory segment is calculated independently by each process. The algorithm that determines this key is: key = base_number + (10 * sl) + il, where base_number is fixed for all session levels and both sl and il are defined by the process's sl and il.
    - Within the shared memory segment, this call creates and initializes a listen queue and an array of buffers. The size of the array is currently a design parameter of value five.
    - The indices of the buffer array are used as the pseudo-socket descriptors.
- bind –ensures the pseudo-socket descriptor that is passed in corresponds with the listen queue socket descriptor
- listen –ensures the pseudo-socket descriptor that is passed in corresponds with the listen queue socket descriptor
- accept – blocks until a connection request (pseudo-socket descriptor) is placed into the listen queue by the Session Server; extracts the pseudo-socket descriptor and returns it

Because the pseudo-socket functionality was developed to fulfill those requirements expected by the protocol server, an alternate interface was created to allow the Session Server (Socket Relay) to interact with the shared memory structure. The Session Server (Socket Relay) is able to open the shared memory segments by independently calculating the key in the manner described above. However, in order to facilitate the "opening" of a pseudo-socket between the protocol server and the Session Server (Socket Relay), the Session Server (Socket Relay) calls a procedure that searches an array of pseudo-socket connections in shared memory to see if there are any connections free to support the current request. This procedure searches the array of connections to see if any of them do not have

the IN_USE bit set. It returns the first available index number or, if none are available, the "connection" is refused.

Within this stage, there were actually two different phases. The pseudo-socket interface was implemented and tested within a single security level first. This restriction allowed faster debugging of the new code as a Ring 3 process. Once the pseudo-socket interface was fully functional at a single security level, it was extended to support multiple security levels.

IV. CONCLUSIONS

A. COMPARISON WITH OTHER WORK

The pursuit of a cost-effective multilevel secure local area network is not new. Several research projects have been dedicated to finding solutions that are inexpensive and provide the capability for a user to use one terminal to access information at different classification levels or pass information between users at different classification levels. Although the results of the previous research have been implemented and presented as solutions to the MLS LAN problem, we believe that each system still has shortcomings that have been addressed by our proposed solution.

1. The NRL Network Pump

The Naval Research Laboratory (NRL) Network Pump [Ref. 23] was developed to allow messages from a system operating at a low security level to be sent to a system operating at a high security level, but not in the reverse direction. It is designed to provide connectivity between multiple single-level systems at different security levels, resulting in a multiple single-level network. While the multiple single-level (MSL) security architecture approach is better than an air-gap and "sneaker-net" solution, it falls short of the true multilevel secure (MLS) solution.

The NRL Network Pump's developers propose that using "a handful of trusted devices to separate information" leads to a less expensive and shorter evaluation and certification process. What is not mentioned is the cost of maintaining separate local area networks (LANs) to allow the replication of data required by the NRL Network Pump solution. While developing the software design, the developers recognized that acknowledgments from high to low are important, but that they also provide a vehicle for covert channels. To avoid this problem, the NRL Network Pump decouples the acknowledgment stream by using statistically modulated acknowledgements. Another important component of the design is the use of wrappers at the low and high servers. The

wrappers allow the applications to communicate with the pump. In order to prevent an application at the low server from arbitrarily pinging processes on the high server, the developers introduced a pump administrator. The pump administrator was an important addition to the basic NRL Pump model because it provides even more assurance against Trojan Horses.

In contrast to the NRL Network Pump, our MLS LAN proposal does not require the maintenance of multiple single-level LANs, but it would still be capable of providing connectivity between LANs of varying levels. Additionally, the majority of the equipment is COTS-based with the exception of the high assurance server and the trusted computing base extension. The XTS-300 represents an existing resource that the DoD has already invested in, and since it has been evaluated at Class B3 in accordance with the TCSEC, the XTS-300 provides protection against Trojan Horses.

2. NRL's MLS Distributed Computing Infrastructure

Another development that is more recent is presented by Kang, et al [Ref. 24]. In this paper, the authors propose using an NRL Pump object, renamed a "flow controller", to ensure only data authorized by a "policy server" is transmitted from one classification domain to another. The authors propose using cryptographic solutions to provide for secrecy, integrity, and identification between the policy server and the flow controller to provide a high level of assurance that messages from the policy server can not be spoofed (see Figure 23).

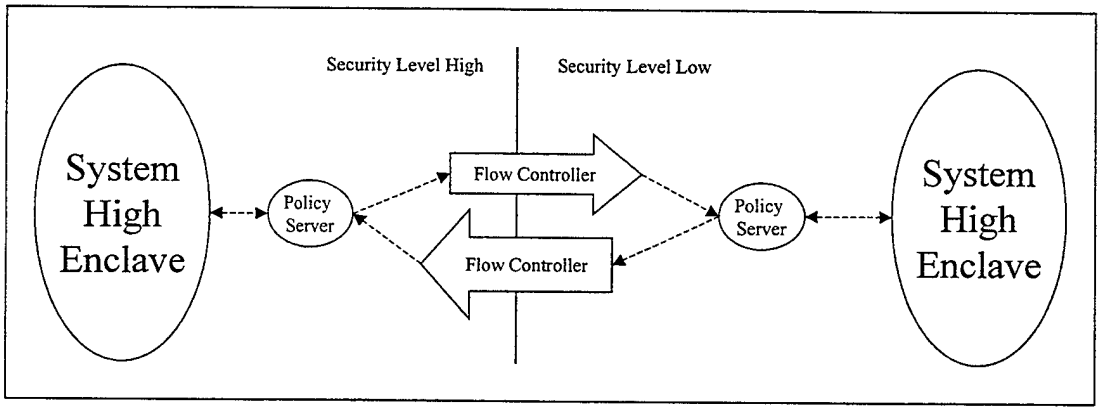

Figure 23. NRL's MLS Distributed Computing Infrastructure

It is assumed that the flow controller can not be bypassed and that only data authorized by the policy server is forwarded to and from the flow controller. The policy server is intended to be a single level platform of "modest trust". The flow controller is built upon a high assurance component, and the system high enclave is a network of arbitrary size functioning in a system high mode. This system high enclave can consist of almost any type of asset (COTS or not).

The primary weakness in the design is that the policy server must be trusted to be both automated and fool proof. If it is not automated, the goal of an interconnected world becomes impossible since the choke point created by a manual policy server would prohibit the exchange of significant amounts of data. This automated policy server is intended to be designed using only "modest trust", but the article does not delineate what criteria are used to determine "modest trust". The effectiveness of a "modest trust" automated policy server (AKA guard) becomes questionable since it must make its release/not release decision based on data content. In a system high enclave all data is assumed to have a virtual label at the maximum sensitivity level of any data stored in the enclave. Therefore, a simple check of the data's label is of no use in improving throughput through the policy server. Trusting an automated "modest trust" policy server to make release decisions based on data content would appear to lead to a slow system with questionable reliability.

Our design overcomes these shortcomings by acknowledging the need to build a secure system based upon a high assurance asset that labels all data. One "expensive" high assurance asset can be used to provide data flow assurances to a large number of low cost clients producing an aggregate low cost solution to the MLS LAN problem. The high assurance server can prevent the exfiltration of data since it was designed from the ground up to do so. The XTS-300, an example of a high assurance server, was designed in accordance with very stringent engineering methodologies and then evaluated by external teams adhering to the tenants of the TCSEC [Ref 2]. This evaluation process adds to the cost of the server, but it provides the assurance required to build a secure network. In our design, we utilize the XTS-300 because it is currently in use in DoD facilities. Using an

existing system leverages previous development expenditures and dramatically reduces the development risk. In contrast, the design and implementation of a policy server has yet to be accomplished, and seems unlikely considering the nearly unlimited ability of a Trojan Horse application to hide sensitive data in seemingly innocuous documents. Our design greatly limits the threat of a Trojan Horse exfiltrating data by requiring labels on all data and preventing the downgrading of data with out human intervention.

B. RECOMMENDATIONS FOR FUTURE RESEARCH

1. Secure LAN Server

In future iterations, of the Secure LAN Server demonstration, integration with inetd as a daemon would be desirable from a consistency standpoint. All other servers are registered with inetd, the Secure LAN Server should be no exception. However, the benefits of inetd such as limiting the number of server processes running at all times and simplifying the coding of the individual server applications were not significant factors for our proof of concept demonstration. The Secure LAN Server only adds one extra process to the system and the coding was largely extracted from Stevens [Ref. 22], hence requiring little effort on our part. Once the application is installed as a trusted daemon, it will not be possible to view run time debug statements. This implies the log file functionality will need to be designed and implemented prior to converting the Secure LAN Server to a daemon.

2. Connection Database

For demonstration purposes, the CDB was designed to allow the system administrator to make additions, deletions, and modifications to the CDB records by directly modifying the initialization file. At TPS startup the initialization file is read into RAM and used for all hardware identification and authentication decisions. Currently, there are no provisions for modifying the image of the CDB in RAM. This design decision requires that the system administrator restart the TPS after each set of CDB modifications. Future work on the Connection Database Module should allow the system administrator to modify the RAM image of the CDB and have those modifications be reflected in the CDB

initialization file. Since, the TPS will be compiled as a daemon in the future the CDB interface will most likely have to be provided via a separate process. The CDB update process could communicate with the TPS via the network's trusted path or via some additional interface local to the server platform. Utilizing the existing network trusted path functionality with a conditional branch if CDB update functionality is desired might prove easier and more flexible to implement.

We believe that restarting the protocol servers every time a configuration change is made would be detrimental to the purpose of this software application. Protocol servers should have high rates of availability, and requiring the server to be shutdown and restarted every time the system administrator has to add TCBE clients to the Secure LAN Server would be counterproductive. Providing a real-time update capability to the CDB and restricting the use of that capability to the system administrator on the high assurance server should provide sufficient assurance that unauthorized modifications will not take place.

3. Trusted Path Server

The Trusted Path Server (TPS) is currently a program that runs from the command line. This implementation was used to simplify the modification-to-test cycle. A future implementation of the TPS program as a daemon that is initialized when the XTS-300 is turned on or rebooted would be more appropriate for the final fielded product, but would have slowed the development process.

This daemon could function in response to inetd connection requests and provide inetd functionality to the various protocol servers. It would be best if the Secure LAN Server provided the inetd "wake-up call" functionality to any desired protocol server not already running. One of the constraints of the current implementation is that the protocol server actually creates the shared memory used for communicating between the Secure LAN Server and itself by a call to pskt:socket(). The protocol server eventually blocks on the listen queue via a call to pskt:accept(). Then the Session Server uses the listen queue initialized by the pskt:socket() call to awaken and to pass the pseudo-socket identifier to the protocol server. If the protocol server is not currently running, this means that the "wake-up

call" must be passed via some other mechanism – a call to load_process() perhaps. At this point, the Session Server must either create the shared memory segment itself, or wait until the protocol server creates and initializes the shared memory segment, before attempting to place the connection id in the listen queue.

The first case, where the protocol server is already running, is currently handled and considered sufficient for our proof of concept demonstration. Implementing a Session Server that is capable of starting the protocol server has been left as future work, but should be fairly simple to implement since the available get_shared_memory() system call already handles the case where the shared memory segment exists. The system does not attempt to recreate the shared memory segment, but simply returns a pointer and the shared memory identifier associated with the existing segment.

4. Hardware Identification and Authentication

Hardware identification and authentication, although vital to the implementation of a true trusted path, was stubbed out for demonstration purposes. Although the hardware identification number was verified against those that were contained in the Connection Database, no authentication was actually performed. Our design envisions the use of public-key cryptography as a possible authentication solution and the rest of our design reflects this.

Public-key cryptography can be used to simultaneously protect the secrecy of the TCBE hardware ID included in the SAS and provide authentication between the TCBE and the Secure LAN Server. The TCBE signs the SAS using its private key and then uses the Secure LAN Server's public-key to encrypt the SAS before it sends it over the network. There are several well-documented public-key identification and authentication algorithms available. Selection and implementation of a suitable algorithm has been left as future work.

5. Negotiate Session Key

One of the simplifying assumptions of this thesis is that the LAN is physically protected against eavesdropping or interceptions. Consequently, the design did not implement session key negotiation. We recognize that, for unprotected networks, there is a

requirement for session encryption, so we have included a possible design solution that uses symmetric-key encryption.

Symmetric-key encryption was chosen over public-key encryption because it is generally faster and thus more appropriate for bulk data encryption. Additionally, choosing symmetric-key encryption allows for the future inclusion of any number of symmetric-key hardware solutions that would yield even greater performance. A public-key exchange algorithm, Oakley [Ref. 5], was chosen as the method for calculating the one-time session key for symmetric encryption since it does not require the session key to be transmitted over the network. Not actually transmitting the key over the network prevents the cascading compromise of future communications that could occur if symmetric session keys were used to protect all data flow and future in-band key updates. The primary advantage to using public-key cryptography is that it allows the re-keying of clients using the Secure LAN in-band communications to transmit the new public keys. This is possible since individual hardware components can recalculate a new key pair. The key update is completed when a new public key is passed to the other necessary parties. All of this can be accomplished without ever passing the private keys in-band. One of the first rules to follow when developing an encryption system is to avoid passing private or any symmetric keys in-band since it minimizes the vulnerability of the system.

After key exchange, a test message should be sent as an automatic communications check to ensure that both the TCBE and the TPS correctly calculated the one-time session key. The test message can be very simple and random. We depend on the diffusion properties of the encryption algorithm to guarantee that only a valid key will yield a correct result when cipher text is decrypted. As long as the randomness of the message fits some predetermined format, it should be possible for the endpoints to determine if they have a correct connection without introducing a vulnerability to a known plain text attack. If the TPS is unable to decrypt the test message, then the connection is terminated. If the TPS can decrypt the test message, the user begins identification and authentication procedures. A

comparison of alternate public-key exchange algorithms and final implementation on the XTS-300 remains to be done.

6. User Identification and Authentication

For demonstration purposes, the user identification and authentication functionality has been emulated and not fully implemented. We only require that the TCBE provide a valid hardware identifier and the user enter the security level and integrity level of an existing protocol server to establish communications. Anything that is entered for user name and password is accepted as valid by the program. The final product should either interface directly with the STOP databases or use an interface provided by Wang Government Services, Inc. such as the pseudo-terminal.

Several factors have to be taken into consideration when deciding whether to use the pseudo-terminal provided by the XTS-300 to accept user and session information. Although we did not research this area in depth, there is a concern that should be mentioned. The establishment of a trusted path for the Secure LAN Server depends on being able to "intercept" communications between the server's TCB and the TCBE and "wrap" the data with the appropriate encryption techniques. Instead of having the user interface directly with the pseudo-terminal, it may be easier to get the information from the user over the trusted path and then pass the login information to the kernel via the pseudo-terminal identifier. The pseudo-terminal would have to remain active for the duration of the user's session to ensure that the audit trail has accurate information.

It would be more efficient if the final product were able to make use of direct calls to the STOP databases since this would alleviate the need to start yet another set of processes supporting a pseudo-terminal used only for login. Since the login code is currently only available to the XTS-300 session server, this implementation would require modification to existing source code. We believe either implementation would work and the tradeoffs are implementation time versus performance.

7. Trusted Prompt

Our trusted prompt, which is used for all valid SASs except the initial login SAS, supports a subset of the usual trusted path functions; the user may continue or logout. The restrictions placed on the commands permitted over the Ethernet trusted path prevent administrative functions from being performed from client PCs. This feature reduces the possibility that the system could be subverted by a user from an external location. However, the final system might extend the number of commands that can be supported from a TCBE to allow functions such as changing user password, which will require an interface to the STOP 4.4.2 User Access Databases.

8. Multiple Protocol Support

For demonstration purposes, our thesis implements a Session Relay that supports one protocol, but the design can be extended to multiplex between multiple protocols. Protocol differentiation would be accomplished, in the normal manner, by assigning different ports to each protocol. The Secure LAN Server would then use select() functionality to multiplex the various LAN socket connections. This would not be difficult, but would require a restructuring of the shared memory structure used to communicate between the protocol server and the Secure LAN Server. Multiple listen queues would be added, one for each protocol server.

9. Shared Memory Structure

Some protocol servers function by accepting a connection identifier, creating a child to handle the request, then closing the connection identifier so that only the child has the connection open. This works because the child process inherits all connection identifiers from the parent. In order to avoid marking the pseudo-socket as not in use, there must be a mechanism to map child processes to pseudo-sockets that is integrated with the pseudo-socket my_close() function. The my_close() call should only mark the pseudo-socket as not in use if there are no processes, parent or child, with the connection open. A list of process identifiers associated with each pseudo-socket would be sufficient to keep track of open

connections, but this method raises questions such as 1) how to know when a process unexpectedly quits and 2) how to know the child has been created. Time stamping each access to the pseudo-socket may provide the answer. A pseudo-socket could be considered stale after some arbitrary time, such as ten minutes, had elapsed without activity. It would be assumed the protocol server's child process would make some call using the pseudo-socket identifier within a ten-minute time frame. When the child does make the call, the child's PID would need to be mapped to the pseudo-socket as opening the pseudo-socket. This time out is similar to the functionality provided by many other network protocols such as ftp, point-to-point protocol, mail servers, et cetera.

Additionally, the listen queue should be re-implemented as another pseudo-socket that allows blocking. This would simplify the handling of multiple socket() calls by a protocol server. Currently the listen queue descriptor is set to a fixed value since there is at most one listen queue in the shared memory structure. This fixed value was arbitrarily set at one greater than the maximum pseudo-socket identifier. Having one listen queue limits each protocol server to only one socket/bind/listen sequence. This unnecessary limitation can be corrected by converting the listen queue to a pseudo-socket. Once a correct version of select() is implemented using signals, this conversion should be straightforward. Select() can then be used inside accept() to force proper blocking behavior.

De-allocation of the shared memory segment also needs to be addressed. Since the current implementation assumes that protocol servers are started ahead of time, there is no real reason to de-allocate the shared memory segments. However, in a future implementation where the protocol servers are started on the fly at whatever level is necessary, it will become necessary to return the scarce shared memory resources to the system. The best time to de-allocate the shared memory might be when the protocol server closes its listen pseudo-socket descriptor, or possibly its last listen pseudo-socket descriptor, assuming that a protocol server that is no longer listening is about to shutdown. The real issue is to design this in such a way that the protocol server need only make normal socket

calls of the pseudo-socket interface that, in turn, initiates the de-allocation of shared memory.

10. Improving Through-put

Currently the pseudo-socket select call is just a busy loop, with an included sleep call to avoid slowing the overall system performance down excessively. A mechanism needs to be added to allow the select call to block for an arbitrary time, from zero seconds to indefinitely. Blocking indefinitely could be implemented by using a counting semaphore. The counting semaphore would be assigned a set of sockets that it would keep track of the number of bytes available to a protocol server; when its value is greater than zero, then select() should stop blocking and indicate which sockets in the set have data available.

Using poll() without a call to sleep would easily simulate not blocking at all. However if we wish to block for a maximum time, but return as soon as data is available, we need to implement some form of signal communication between the Session Server and the pseudo-socket select() call. The best method for implementation would require the select() call to check whether there is data currently available; if not, then select() would block on a *data available* signal from the Session Server. The Session Server could generate this signal every time it calls xfer_skt_buff() (transfer data from socket to shared memory buffer).

Likewise, a similar signal mechanism needs to be implemented for data flow from the pseudo-socket write to the Session Server. This becomes more complicated on the Session Server side because the Session Server needs to be able to block on TCP/IP sockets as well as on data available in the shared memory buffer. Careful attention is required to ensure that data available signals from the pseudo-socket do not interfere with communication on the TCP/IP sockets. It may even make sense to have the Session Server fork itself into two processes, one for inbound data and one for outbound data. There are several design changes required to improve overall throughput.

11. Protocol Server Integration

The first demonstration linked an echo server to our pseudo-socket library. This proved that the data flow works as expected and that it is easy to link an existing TCP/IP application to our pseudo-socket implementation. Future students need to link the IMAP server, recently ported by Eads [Ref. 25] to the XTS-300, to the pseudo-socket library and expand the pseudo-socket interface if necessary. Currently fctl() and ioctl() are not provided and may be needed to correctly mimic the expected socket interface. The ported echo server only required the function calls: socket, bind, listen, accept, select, read and write.

C. CONCLUSIONS

The Multilevel Secure Local Area Network (MLS LAN) presented in this thesis is intended to utilize COTS clients and existing multilevel high assurance hardware to allow single level clients access to multilevel data. We propose a design and provide a proof of concept for the implementation of the interface between a trusted computing base extension (TCBE) and a protocol server executing in a single level on the XTS-300. This interface includes procedures to create a network trusted path between a TCBE and the Secure LAN Server; utilize the trusted path for user identification and authentication; then act as a trusted relay between the protocol server and the TCBE. All transmitted data has the potential to be protected by encryption to provide assurance as to the integrity and confidentiality of the data if it is passed over an unprotected LAN.

We have proven the feasibility of implementing a Secure LAN Server on Wang Government Service's XTS-300 while preserving the potential for a future evaluation at Class B3 following the guidance contained in the TNI [Ref 7] of the TCSEC [Ref 2] or an equivalent Common Criteria profile. This proof of concept demonstration mitigates much of the risk in moving towards a full scale Multilevel Secure LAN. Coupled with the work accomplished by Irvine, et al. [Ref. 4], Eads [Ref. 25], and ongoing research at the Naval Postgraduate School into the feasibility of creating a high assurance TCBE, we have made considerable strides towards providing a cost effective solution that takes advantage of

COTS software and hardware while still providing secure access to multilevel data in a network environment.

APPENDIX A. SECURE LOCAL AREA NETWORK SERVER SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

1.0 Introduction

1.1 Purpose

The concept of a secure session has traditionally been constrained to sessions that are on a high assurance workstation or established via a point-to-point connection. With a point-to-point link, a special out-of-band sequence can be defined that ensures trusted path initiation. An Ethernet network does not provide a point-to-point connection; there is no guarantee that when the connection is established, the user is connected with the desired host. [Ref. 26: p. 1]

The purpose of this document is to define the software requirements that will establish a trusted path and a secure session between a multilevel, high assurance server and a trusted computing base extension (TCBE) over an Ethernet network. Assurance and security requirements outlined by the TCSEC [Ref. 2] will be incorporated into these software requirements.

1.2 Scope

The product, the Secure LAN Server, is designed to interface with STOP 4.4.2, the operating system associated with the XTS-300 high assurance workstation produced by WANG Federal, Inc. This specification is intended to form the basis for the design of a software product that will support the establishment of a secure session using a trusted path across an untrusted local area network. The software product can be separated into two areas of functionality: the Trusted Path Server and the Session Server.

1.3 Glossary of Abbreviations and Definitions

See Appendix F of "Secure Local Area Network Services For A High Assurance Multilevel Network " by Lieutenants Susan BryerJoyner and Scott D. Heller.

1.4 References

See List of References from "Secure Local Area Network Services For A High Assurance Multilevel Network " by Lieutenants Susan BryerJoyner and Scott D. Heller.

2.0 General Characteristics

2.1 Introduction

The Secure LAN Server is part of the Multi-level Secure (MLS) Local Area Network (LAN) development project sponsored by Dr. Cynthia Irvine and the Center for INFOSEC Studies and Research at the Naval Postgraduate School. Technical support and access to proprietary source code was provided by WANG Federal, Inc.

2.2 Product Perspective

The Secure LAN Server requires the trusted computing base extension (TCBE) card (to be developed in another thesis). The Secure LAN Server interfaces with the STOP 4.4.2 operating system associated with the XTS-300 high assurance server produced by WANG Federal, Inc. The high-level system diagram is shown in Figure 1.

2.3 Product Functions

The Secure LAN Server is expected to provide the following functions to establish a trusted path, establish and maintain a session in a manner that preserves a secure state on the server, and ensure enforcement of the access control policy for client requests.

- establish a trusted path using a cryptographic algorithm, or combination of algorithms, that provides authentication and secrecy for data transmitted over the LAN.
- accept user identification and authentication information over a trusted path established between the Trusted Path Server and a trusted computing base extension (TCBE) and verify the user information against that contained in the STOP 4.4.2 User Access Databases

- establish a session on the server at the default security and integrity level for each valid user
- provide a pseudo-socket to which the protocol server can connect with minimal modification of the protocol server
- provide trusted commands **continue** and **logout**

The Secure LAN Server is expected to emulate, to the greatest extent possible, the XTS-300 login and trusted prompt interface.

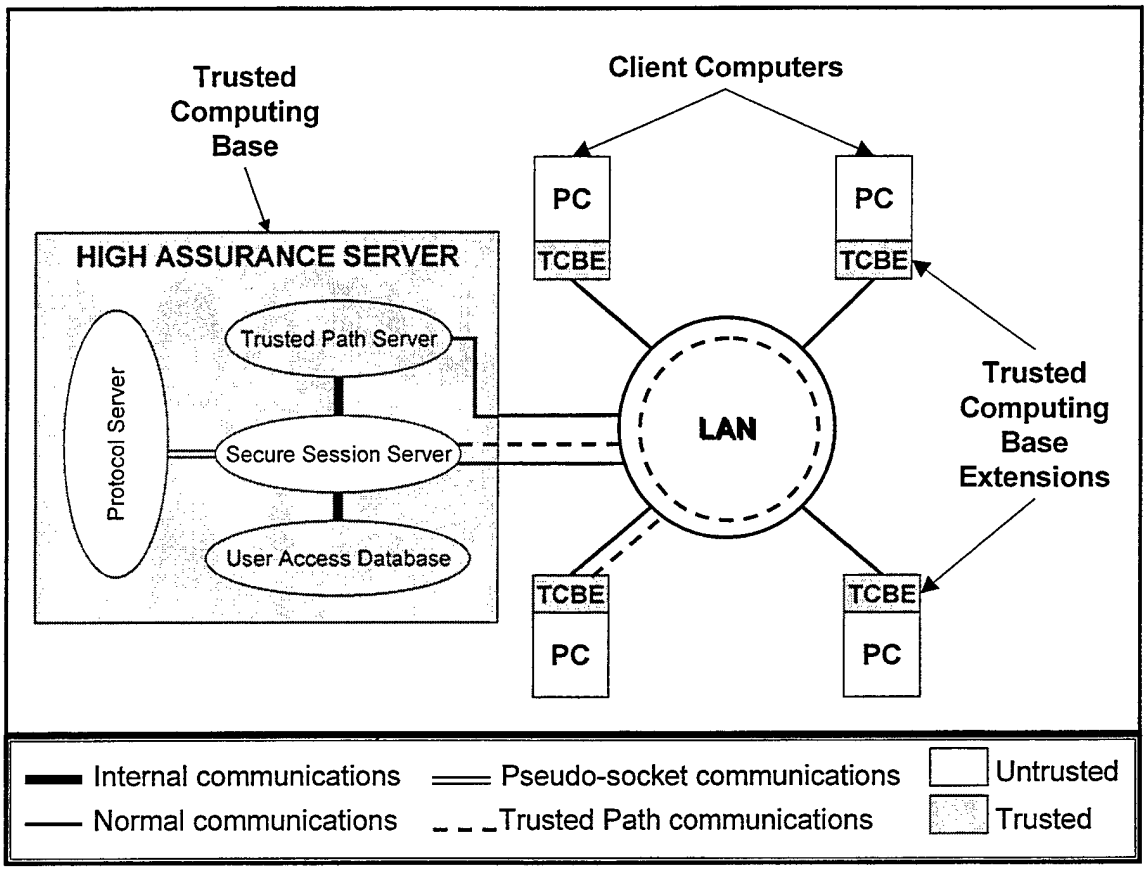

Figure 1. Multilevel Secure Local Area Network

2.4 User Characteristics

Users are expected to be computer literate and familiar with handling information at different security levels in accordance with any applicable security policies.

2.5 General Constraints

- The Secure LAN Server software shall execute under release 4.4.2 of the STOP operating system.
- The Secure LAN Server software shall be written in C.
- The Secure LAN Server shall run on the XTS-300.
- The Secure LAN Server shall be designed to interface with the Trusted Computing Base Extension (TCBE).
- The product is designed so that a network that incorporates it may be evaluated at Class B3 in accordance with the Trusted Network Interpretation of the TCSEC [Refs. 2 and 7].
- The permitted session level at the PC may be constrained by the security level of its physical environment; the permitted session levels may be a subset of the full range of valid security and integrity levels.
- Each TCBE can support only one secure session at any one time.
- The number of concurrent secure sessions a user may have (at multiple PCs) may be restricted by the system administrator.
- Amount of trusted code will be kept to a minimum.

The Secure LAN Server is a key subsystem. If the Secure LAN Server malfunctions, it shall not cause the STOP 4.4.2 operating system to crash or hang. Availability includes operating software, the application software product, and server hardware.

2.6 Assumptions and Dependencies

- The product is designed to work with the XTS-300 and the TCBE developed as part of the MLS LAN project.
- The TCBE shall have the requisite software and cryptographic keys installed to create the trusted path.

- Protocol servers that support applications on a standard MS Windows NT PC have been ported to the XTS-300.

3.0 Specific Requirements

3.1 Functional Requirements

This section contains the details necessary to create the design specifications of the Secure LAN Server. It is organized in five sections.

3.1.1 Establish Trusted Path

3.1.2 User Login

3.1.3 Establish Session

3.1.4 Trusted Prompt

3.1.5 Pseudo-Socket Interface

Section 3.1.5 is provided to allow the system to emulate socket functions internal to the server. The design of the few socket functions required and implemented is specific to the products developed in this thesis.

3.1.1 Establish Trusted Path

Introduction

This module establishes the trusted path between the user and the trusted computing base of the remote server.

Inputs

- int – socket descriptor

Processing

Receives secure attention sequence (SAS) from the TCBE. Verifies SAS format. Extracts the TCBE hardware identification number and verifies it against the connection database.

Outputs

- int – controlling active process identification (CAPID)

- *int – TCBE hardware identification number

Error Handling

If the SAS is not in the proper format or if the TCBE hardware identification number is not in the connection database, the connection is refused.

3.1.2 User Login

Introduction

This receives user and session information from the TCBE.

Inputs

- int – socket file descriptor of connection

Processing

This module will initially be hard-coded to accept any input as user name and password. While the session security level and integrity level must be in the proper format, the values are not checked against the STOP 4.4.2 User Access Databases. The fully implemented design would use well-defined functions provided by the STOP 4.4.2 operating system to verify the user and session information against that contained in the User Access Databases.

Outputs

- struct – contains fields
    - int – user and session information valid; if valid, TRUE; otherwise, FALSE
    - char[] – contains user's name up to MAX_USER_NAME
    - int – session security level
    - int – session integrity level

Error Handling

None.

3.1.3 Establish Session

Introduction

This module establishes a session on the remote server at the user's desired security and integrity levels.

Inputs

- struct – contains fields
    - int – user and session information valid; if valid, TRUE; otherwise, FALSE.
    - char[] – contains user's name up to MAX_USER_NAME
    - int – session security level
    - int – session integrity level

Processing

This module uses well-defined functions provided by the STOP 4.4.2 operating system to give the Session Server privileges that allow it to communicate between two processes at different mandatory access levels.

Outputs

- int – old privilege set if successful; ERROR_OCCURRED otherwise

Error Handling

If the Trusted Path Server program has not been installed with the proper privilege set, the user will be prompted contact his administrator, who must reinstall the program with the proper privileges.

3.1.4 Trusted Prompt

Introduction

This module provides the trusted commands **continue** and **logout**.

Inputs

None

Processing

This module displays the current session security and integrity levels at the user's terminal, followed by a prompt "Continue or logout?" If the user enters **continue**, the session is reattached in its previous state. If the user enters **logout**, the session is terminated normally.

Outputs

None

Error Handling

Any entries other than **continue** or **logout** will cause the user to be prompted for input again.

3.1.5 Pseudo-Socket Design

Introduction

The pseudo-socket library will provide two interfaces, one to the protocol server and one to the Session Server. Only a few socket calls are required by the protocol server and implemented in this thesis. The design of socket functions is specific to the products developed in this thesis. Some method of synchronization will be required to ensure that both the TPS and the protocol server can access the shared memory segments that will represent pseudo-sockets.

The Session Server (Socket Relay) and the pseudo-socket code will include a base number that will be used to calculate the key value that the TPS and the protocol server will use to access shared memory. One shared memory segment will be created at each security level of the system. The shared memory segments will contain two data structures: a listen queue and an array of connections that each contains a to_server and to_client buffer. The index of the array for a particular connection is used as the pseudo-socket descriptor.

When the protocol server is executed, it calls **socket** to create a pseudo-socket. The modified **socket** call uses the base key value, security level and integrity level of the protocol server to calculate the key required to create and open the shared memory segment

at the proper security level. **Bind** and **listen** are procedures that check to ensure that the pseudo-socket passed in is that for the listening socket. The protocol server then blocks on an **accept** waiting for a connection request on the listen queue in the shared memory segment.

As the TPS accepts connections from TCBEs, it creates Session Servers. The Session Server calculates a key to shared memory in the manner described above and opens the shared memory segment that exists at the desired security level for the session. After receiving a pseudo-socket descriptor in response to a shm_struct connection request, the Session Server must place the pseudo-socket descriptor in the listen queue of the shared memory segment.

When **accept** detects data in the listen queue, it removes the first pseudo-socket descriptor from the listen queue, and returns the pseudo-socket descriptor. At this point, the connection has been established between the Session Server and the protocol server. The protocol server now blocks on **select** while waiting for additional data. Modified **my_read**, **my_write**, and **my_close** are used in succeeding pseudo-socket manipulations.

3.1.5.1 socket

Introduction

Creates a pseudo-socket and returns a pseudo-socket descriptor. The prototype should be int socket(int *domain*, int *type*, int *protocol*).

Inputs

- int – *domain* specifies the communications domain
- int – *type* specifies the type of the socket (SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW)
- int – *protocol* specifies a particular protocol to be used with the socket

Processing

Using libraries developed to emulate normal socket behavior internal to the Secure LAN Server, this function creates a pseudo-socket by creating if necessary and connecting to a shared memory segment that will simulate stream input/output.

Outputs

- int – 0 if *successful*; -1 otherwise

Error Handling

Return –1 if listen queue socket already allocated.

3.1.5.2 bind

Introduction

This function is provided to permit compilation of source code that depends on the modified libraries; currently, pseudo-sockets do not require this function and simply ensure that the socket descriptor presented is the correct one. The prototype should be: int bind(int *sockfd*, struct sockaddr * *name*, int *namelen*).

Inputs

- int – *sockfd specifies* a socket that exists in a name space (address family) but has no name assigned
- struct sockaddr * - *name* specifies the name to be assigned to the socket
- int – *namelen* indicates the length of the name pointed to by **name*

Processing

None.

Outputs

- int – 0 if successful; -1 otherwise

Error Handling

Unsuccessful if attempting to bind to any pseudo-socket other than the listen queue socket returned by a previous call to socket or if the sockfd is not valid in use.

3.1.5.3 listen

Introduction

This function is provided to permit compilation of source code that depends on the modified libraries; currently, pseudo-sockets do not require this function and simply ensure that the socket descriptor presented is the correct one. The prototype should be: int listen(int *sockfd*, int *backlog*).

Inputs

- int – *sockfd* specifies the socket to listen to
- int – *backlog* specifies the maximum length that the queue of pending connections may grow to

Processing

None.

Outputs

- int – 0 if successful; -1 otherwise

Error Handling

Unsuccessful if attempting to listen to any pseudo-socket other than the listen queue socket returned by a previous call to socket or if the sockfd is not valid in use.

3.1.5.4 accept

Introduction

This function provides functionality for pseudo-sockets that is identical to **accept**. The prototype should be:

int accept(int *sockfd*, struct sockaddr **addr*, int **addrlen*).

Inputs

- int – *sockfd* specifies the socket that is listening for a connection
- struct sockaddr * - *addr* is a result parameter that is filled in with the address of the connecting entity as it is known to the communications layer (domain specific format)
- int * - *addrlen* is a value-result parameter that contains the length in bytes of the address returned

Processing

Using libraries developed to emulate normal socket behavior internal to the Secure LAN Server, this function is designed to block until a connection request is made. A connection request is simulated by the placement of a pseudo-socket descriptor into the listen queue located in the shared memory segment at the correct security level. At that time, this function removes the first pseudo-socket descriptor and returns it to the protocol server as the pseudo-socket descriptor.

Outputs

- int – if successful, non-negative integer that is the pseudo-socket descriptor for the accepted socket; -1 otherwise

Error Handling

Unsuccessful if attempting to accept from any pseudo-socket other than the listen queue socket returned by a previous call to socket or if the sockfd is not valid in use.

3.1.5.5 select

This function returns the number of ready descriptors contained in the descriptor sets or –1 if an error occurred. The prototype should be:

int select(int *nfds*, fd_set **readfds*, **writefds*, **exceptfds*, struct timeval **timeout*)

Inputs

- int – *nfds* is the number of bits to be checked in each bit mask that represents a file descriptor

- fd_set – **readfds* contains the addresses of file descriptors to be examined to see if any of their descriptors are ready for reading; can be NULL
- fd_set – **writefds* contains the addresses of file descriptors to be examined to see if any of their descriptors are ready for writing; can be NULL
- fd_set – **exceptfds* contains the addresses of file descriptors to be examined to see if any of their descriptors have an exceptional condition pending; can be NULL
- struct timeval - **timeout* specifies the maximum interval to wait for the selection to complete if it is not a NULL pointer; if it is a NULL pointer, the **select** blocks indefinitely

Processing

The function does not use *nfds* and *timeout*. It should be hard-coded to poll every one seconds to see if any of the descriptors are ready for reading, writing, or have an exceptional condition pending.

Outputs

- int – number of ready descriptors; -1 if error

Error Handling

Returns error if one of the I/O descriptor sets specified an invalid I/O descriptor.

3.1.5.6 my_read

This function provides identical functionality to the system call **read**. The prototype should be: int my_read(int *fd*, char **buff*, int *read_limit*)

Inputs

- int – *fd* is the pseudo-socket connection identification
- char – **buff* contains the location to put character data that is read

- int – *read_limit* is the maximum number of characters to read

Processing

This function attempts to read the specified number of bytes from the connection associated with the given socket descriptor into the buffer pointed to by the given pointer.

Outputs

- int – If successful, returns the number of bytes actually read and placed in the buffer; this number may be less than the specified number of bytes; otherwise –1.

Error Handling

Returns error if the pseudo-socket identifier is not in use.

3.1.5.7 my_write

This function provides identical functionality to the system call **write**. The prototype should be: int my_write(int *fd*, const **buff*, int *nbytes*)

Inputs

- int – *fd* is the pseudo-socket connection identification
- char – **buff* contains the location to read character data from
- int – *nbytes* is the maximum number of characters to write

Processing

This function attempts to write the specified number of bytes from the buffer pointed to by the given pointer into the connection associated with the given socket descriptor.

Outputs

- int – If successful, returns the number of bytes actually written to the connection; this number may be less than the specified number of bytes if there is insufficient room in the connection buffer; otherwise –1.

Error Handling

Returns error if one of the I/O descriptor sets specified an invalid I/O descriptor.

3.1.5.8 my_close

This funtion provides identical functionality to the system call **close**. The prototype should be: int my_close(int *fd*)

Inputs

- int – *fd* is the pseudo-socket connection to be closed

Processing

The function closes the specified pseudo-socket descriptor.

Outputs

- int – If successful, returns 0; -1 if error

Error Handling

None.

3.1.5.9 FD_SET

This function provides identical functionality to the socket call FD_SET. The prototype should be: FD_SET(int *fd*, fd_set **bits*)

Inputs

- int – *fd* is the file descriptor to be included in *bits*
- fd_set – *bits* is the set of flags, one of which should be associated with *fd*

Processing

The function includes the specified file descriptor in the file descriptor set.

Outputs

None.

Error Handling

Returns error if the I/O descriptor specified an invalid I/O descriptor.

3.1.5.10 FD_ZERO

This function provides identical functionality to the socket call **FD_ZERO**. The prototype should be: int FD_ZERO(fd_set **bits*)

Inputs

- fd_set – *bits* is the set of flag bits to be set to zero

Processing

The function initializes the set of flag bits to the null set.

Outputs

None.

Error Handling

None.

3.1.5.11 FD_ISSET

This function provides identical functionality to the socket call **FD_ISSET**. The prototype should be: int FD_ISSET(int *fd*, fd_set **bits*)

Inputs

- int – *fd* is pseudo-socket connection identification
- fd_set – *bits* is the set of flags to test if *fd* is set to true(1)

Processing

This function checks whether the bit associated with *fd* is set in the *bit* set.

Outputs

- int – returns a nonzero if *fd* is a member of *bits*, a zero otherwise

Error Handling

Returns error if the I/O descriptor specified an invalid I/O descriptor.

3.1.5.12 FD_CLEAR

This function provides identical functionality to the socket call **FD_CLEAR**. The prototype should be: int FD_CLEAR(int *fd*,, fd_set **bits*)

Inputs

- int – *fd* is pseudo-socket connection identification
- fd_set – *bits* is the set of flags of which the bit associated with *fd* will be set to zero

Processing

The function removes *fd* from *bits* by clearing the bit associated with *fd*.

Outputs

None.

Error Handling

Returns error if the I/O descriptor specified an invalid I/O descriptor.

3.2 External Interface Requirements

3.2.1 User Interfaces

- It should be possible for users described in section 2.4 to use the program by following information provided on the screen and in the Secure LAN Server user manual. Assistance in changing session level (**sl**) will not be provided interactively.
- The program shall use command line prompts to allow the user to enter information. Available options will not be provided on the screen, but will be available in the Secure LAN Server user manual.

3.2.2 Hardware Interfaces

- It shall not be necessary to amend or reconfigure the server to install the Secure LAN Server.
- The screen (hardware and software support) shall be capable of displaying command line prompts.

3.2.3 Software Interfaces

- The program shall run under STOP 4.4.2.
- In order to use the program, the user must be properly authenticated to the system.
- It shall be the responsibility of the system administrator to establish the necessary access rights.

3.3 Performance Requirements

- There will be a maximum of 15 seconds between a user initiating the login process and the system making a visible response.

3.4 Design Constraints

3.4.1 Standard Compliance

Design and development shall conform to Wang Government Services, Inc.'s software development standard.

3.4.2 Hardware Limitations

The XTS-300 characteristics and limitations are as described in Reference 27.

APPENDIX B. SECURE LOCAL AREA NETWORK SERVER SOFTWARE DESIGN SPECIFICATION (SDS)

1.0 Introduction

Refer to Section 1.0 of the Secure LAN Server Software Requirements Specification.

2.0 System Overview

2.1 Introduction

Refer to Section 2.1 of the Secure LAN Server Software Requirements Specification.

2.2 Product Perspective

Refer to Section 2.2 of the Secure LAN Server Software Requirements Specification.

2.3 Product Functions

Refer to Section 2.3 of the Secure LAN Server Software Requirements Specification.

3.0 Design Considerations

3.1 Assumptions and Dependencies

Refer to Section 2.6 of the Secure LAN Server Software Requirements Specification.

3.2 General Constraints

Refer to Section 2.5 of the Secure LAN Server Software Requirements Specification.

3.3 Development Methods

Refer to Chapter II, Section B in "Secure Local Area Network Services For A High Assurance Multilevel Network by Lieutenants Susan BryerJoyner and Scott D. Heller.

4.0 Architectural Strategies

Refer to Chapter II – Section C and Chapter III in "Secure Local Area Network Services For A High Assurance Multilevel Network by Lieutenants Susan BryerJoyner and Scott D. Heller.

5.0 System Architecture

Refer to Chapter III in "Secure Local Area Network Services For A High Assurance Multilevel Network by Lieutenants Susan BryerJoyner and Scott D. Heller.

6.0 Module Design Overview

The module design overview is intended for use with the source code to better understand the current state of the Secure LAN Server design and implementation. Each of the following sub-sections details a module in our final design and corresponds to a source file. The design of each module is intended conform, as close as possible given the limitations of the C programming language, to our object oriented model of the Secure LAN Server. The interface/exports section is used to provide a listing of each function provided by a module. For a detailed description of each function, refer to the appropriate section of Appendix C.

6.1 Buffer I/O

Classification

Data Store Module

Definition

Provide circular queue buffer for network I/O.

Responsibilities

Prevent buffer overflow while allowing controlled reads from socket IO.

Constraints

Buffer size is limited to INBUFFSIZE. Currently set to 4096 bytes.

Composition

None.

Uses/Interactions

Uses socket I/O for reading.

Resources

Refer to "buff_io.h" in Appendix C for the #includes.

Each instantiation of a buffer I/O object contains an object of type in_buff_struct. The struct is used to store the circular queue representation in a flat array.

```
struct in_buff_struct
{
      char in_buff[INBUFFSIZE];
      int  read_idx;
      int  write_idx;
};
```

Processing

For all operations a read and write index are used to maintain the next item to read and the next location to write to. Data is stored in an array of char. Read and write indices are manipulated to force circular queue behavior, by using modular arithmetic to limit the range of each index to [0,INBUFFSIZE]. When reading the read_idx may never pass the value of write_idx. When writing write_idx may never pass the value of read_idx. For purposes of the circular queue, "pass" means to be incremented when equal to the other index. When the two indices are equal the queue is empty.

Interface/Exports

Refer to "buff_io.h" in Appendix C for a full description of each function.

```
void init_buffer( struct in_buff_struct * this );
int poll_ok_to_read( int fd );
int poll_ok_to_read_block( int fd, int milliseconds );
int poll_ok_to_write( int fd );
char * get_token( int fd, struct in_buff_struct *queue,
                  const char delim, int nbytes );
char *empty_buff(struct in_buff_struct *queue);
int buff_io_read(struct in_buff_struct * this, char * data, int n);
int num_char(struct in_buff_struct *queue );
char peek_char( struct in_buff_struct *queue);
char remove_char(struct in_buff_struct *queue);
int bytes_free( struct in_buff_struct * queue);
int empty( struct in_buff_struct *queue );
void add_data( const char *data, struct in_buff_struct *queue,
               int num_read );
int add_data_part( const char *data, struct in_buff_struct *queue,
                   int num_read );
int get_data(int fd, struct in_buff_struct *queue );
void print_buff_queue( struct in_buff_struct *this );
```

6.2 Connection Database

Classification
Data Store Module
Definition
Provide interface to Connection Database, to include initialization and modification of the controlling active process. The Connection Database is initialized from an initialization file and maintained in memory during run time.
Responsibilities
Provide basic database functionality for connection database. Functions needed include record retrieval by hardware ID and controlling active process field modification of a record identified by a hardware ID. Used to determine if a TCBE has an active connection and which Session Server is the Controlling Active Process (CAPID).
Constraints
Maximum number of records is defined by MAX_CLIENT. (10 records). Maximum record length is defined by MAX_RECORD_LEN. (20 bytes). Hardware ID is 1-3 digits. Public-key. The remaining bytes, however it is currently not used. Controlling active process is only maintained in the memory version of the database. It is the result of a call to getpid() – an int.
Composition
None.
Uses/Interactions
Uses socket I/O for reading.
Resources
Refer to "cdb.h" in Appendix C for the #includes. The Connection Database module uses an array of cdb_records to store the

Connection Database in memory.

```
struct cdb_record
{
   unsigned hw_id;
   unsigned public_key;
   unsigned capid;
};
```

Processing

Hardware ID lookups are currently done by exhaustive search. O(n).

Interface/Exports

Refer to "cdb.h" in Appendix C for a full description of each function.

```
int init_cdb( );
int update_CDB( int hw_id, int new_CAP );
int get_CAPID( int hw_id );
void print_cdb_record(struct cdb_record * this_record );
void print_cdb();
```

6.3 IO Utilities

Classification

Wrapper

Definition

Provide interface to system socket I/O.

Responsibilities

Provide robust write function for socket IO.

Constraints

None.

Composition

None.

Uses/Interactions
Uses socket I/O for writing.
Resources
Refer to "io_util.h" in Appendix C for the #includes.
Processing
Returns zero upon successful write. Uses <stdio.h> write().
Interface/Exports
Refer to "io_util.h" in Appendix C for a full description of each function. `int Writen(int fd, void *ptr, size_t nbytes);`

6.4 Listen Queue

Classification
Data Store Module
Definition
Provide FIFO Queue for new connection requests. Uses circular queue structure to avoid buffer overflow condition.
Responsibilities
Provide ability for a calling function to block until data are available. Accept pseudo-socket connection request identifiers.(type int) Deliver pseudo-socket identifier to calling function upon exit.
Constraints
Buffer size is limited to MAX_LQ_SIZE. Currently set to 10 bytes. Base counting semaphore key is LISTEN_Q_SEM_KEY. Currently 5000. Base must be a multiple of 100 to allow calculation of sl an il from the respective level key in msem:sem_open.
Composition
"msem.h" for semaphore used to count data items in listen queue.

Uses/Interactions

Refer to "buff_io.h" in Appendix C for the #includes.

msem:sem_op() as counting semaphore to keep track of the number of items in the queue.

priv_util:calc_key().

Resources

The listen queue module uses the following listen_q_struct to simulate a listen socket queue. The queue uses a similar construct to the buff_io queue, but is intended to hold integers vice bytes.

```
struct listen_q_struct {
      int   initialized;  //according to ANSI C static var
                          // init to zero.
      int   write_idx;
      int   read_idx;
      int   in_buff[MAX_LQ_SIZE]; // pseudo-socket identifiers.
      int   listen_q_sem;         // semaphore id used to block on lq
      key_t listen_q_sem_key;     // key value.
};
```

Processing

When adding an item the listen_q_sem is incremented by 1.

When removing an item sem_op(listen_q_sem, -1) is called to block until some data is available. When the semaphore's value is greater than zero the block is removed and the value of the data is returned.

Interface/Exports

Refer to "listenq.h" in Appendix C for a full description of each function.

```
void lq_init(struct listen_q_struct * this );
void lq_print(struct listen_q_struct * this );
int lq_add_item(int data, struct listen_q_struct *queue );
int lq_empty(struct listen_q_struct * queue );
int lq_num_free(struct listen_q_struct * queue );
int lq_remove_item(struct listen_q_struct * queue );
int lq_peek_item(struct listen_q_struct *queue );
int lq_num_items(struct listen_q_struct *queue);
```

6.5 Semaphore Array

Classification
Wrapper
Definition
Provide simplified interface to arrays of semaphores.
Responsibilities
Allow for the initialization, destruction, and manipulation of arrays of semaphores.
Constraints
MAX_SEMAPHORES is 20 MAX_SEMAPHORES_PER_SET is 20. MAX_SEMAPHORE_VALUE is 32767 Maximum semaphores per array 18. (20 – 2 control semaphores per set)
Composition
Refer to "msem.h" in Appendix C.
Uses/Interactions
Refer to "msem.h" in Appendix C for the #includes. In sem_create and sem_open when used in a ring 2 application the key is used to calculate the desired (sl, il) pair IAW the following formula: key = base + sl *10 + il; This requires the base key be a multiple of 100.
Resources
Refer to "msem.h" in Appendix C for a detailed discussion of the data structures used.
Processing
Refer to "msem.c" in Appendix C.

Interface/Exports

Refer to "msem.h" in Appendix C for a full description of each function.
`int sem_create( key_t key, int initial, int num_sems );`
`int sem_open ( key_t key);`
`void sem_wait(int id, int idx);`
`void sem_signal(int id, int idx);`
`void sem_op(int id, int idx, int value);`
`void sem_close(int);`
`void sem_rm(int);`

6.6 Privilege Utilities

Classification
Wrapper
Definition
Provide simplified to the STOP operating system set privilege calls.
Responsibilities
Enable a predetermined set of privileges needed to acquire exemption from MAC constraints.
Constraints
None.
Composition
Refer to "priv_util.h" in Appendix C.
Uses/Interactions
Refer to "priv_util.h" in Appendix C for the #includes. In sem_create and sem_open when used in a ring 2 application the key is used to calculate the desired (sl, il) pair IAW the following formula: key = base + sl *10 + il; This requires the base key be a multiple of 100.

Resources
Refer to "priv_util.h" in Appendix C for a detailed discussion of the data structures used.
Processing

The following privileges are assigned upon calling enable_priv():
`SIMPLE_SECURITY_EXEMPT;`
`SIMPLE_INTEGRITY_EXEMPT;`
`SECURITY_STAR_PROPERTY_EXEMPT;`
`INTEGRITY_STAR_PROPERTY_EXEMPT;`
Currently no processing is done for the special cases where only a subset of the privileges is required.

Interface/Exports

Refer to "priv_util.h" in Appendix C for a full description of each function.
`ushort enable_priv();`
`void set_priv(ushort priv );`
`int get_current_level(struct level_struct * lvl );`
`key_t calc_key(int base );`

6.7 Shared Memory Module

Classification
Wrapper
Definition
Provide simplified interface to system shared memory.
Responsibilities
Allow for the initialization, destruction, and manipulation of shared memory segments.
Constraints
None.
Composition
Refer to "shm.h" in Appendix C.

Uses/Interactions
Refer to "shm.h" in Appendix C for the #includes.
Resources
Refer to Reference 28, p. 50, p. 10
Processing
Refer to "shm.c" in Appendix C.
Interface/Exports

Refer to "shm.h" in Appendix C for a full description of each function.
`int get_shm(key_t key, void ** addr, size_t size );`
`void *attach_shm( int shmid );`
`void remove_shm( int shm_id, void * addr );`

6.8 Shared Memory Structure

Classification
Data Store Object
Definition
Provide interface to shared memory segment used to pass data to/from the Session Server and the Protocol Server.
Responsibilities
Allow for the initialization, destruction, and manipulation of a shared memory structure stored in shared memory.
Constraints
MAX_OPEN_CONN represents the maximum number of connection buffers available for use as pseudo-sockets in each shared memory structure defined (Currently 5). Only one listen queue may be defined per shared memory structure. Since there is only one shared memory structure per level this, currently, implies only one protocol server listening to one socket per (sl,il) pair.

Composition

Refer to "shm_struct.h" in Appendix C.

Other objects used: listen queue, shared memory, semaphore arrays, and buffer I/O.

Uses/Interactions

Refer to "shm_struct.h" in Appendix C for the #includes.

Resources

```
// Each p_socket connection needs an inbound and outbound
// buffer as well as a flag to indicate if in_use or not.
// The addr should eventually be filled in by the SSS
// and returned in accept().
struct connect_struct {
       int in_use;
       struct in_buff_struct to_svr_buff;
       struct in_buff_struct to_cli_buff;
       struct sockaddr addr; // client address storage
};

// The entire contents of each level's shared memory segment
// lq is used to block on by accept.
struct shm_hdr {
       struct listen_q_struct lq;
       struct connect_struct conn[MAX_OPEN_CONN];
       int    shm_hdr_shmid; // needed for ss_cleanup call to rm shm.
       int    conn_semid;
};
```

Processing

Refer to "shm_struct.c" in Appendix C.

Interface/Exports

Refer to "shm_struct.h" in Appendix C for a full description of each function.

```
int init_shm_hdr( struct shm_hdr ** shm_ptr );
void ss_cleanup( struct shm_hdr * shm_ptr );
int ss_get_hdr(struct shm_hdr ** shm_hdr,
               struct user_ia_struct * ia_data);
void ss_detach_hdr( struct shm_hdr * shm_hdr );
int ss_read(int fd, struct shm_hdr * shm_hdr, char *buff,
            int nbytes );
int ss_write(int fd, struct shm_hdr *shm_hdr,
             const char *data, int nbytes );
int ss_read_fm_svr(int fd, struct shm_hdr * shm_hdr, char *buff,
                   int nbytes );
int ss_write_to_svr(int fd, struct shm_hdr *shm_hdr,
                    const char *data, int nbytes);
void ss_close(int fd, struct shm_hdr *shm_hdr );
int ss_data_avail(  int idx, struct shm_hdr *shmhdr );
int ss_space_avail( int idx, struct shm_hdr *shmhdr );
int ss_socket_error(int idx, struct shm_hdr *shmhdr );
int ss_block_on_lq( struct shm_hdr *shmhdr );
void ss_copy_cli_buff( struct shm_hdr *shmhdr, int idx,
                       struct in_buff_struct *from );
int ss_xfer_skt_buff( struct shm_hdr *shmhdr, int pskfd,
                      int sockfd );
int ss_xfer_buff_skt( struct shm_hdr *shmhdr, int pskfd,
                      int sockfd );
int ss_request_connection( struct shm_hdr *shmhdr );
```

6.9 User Identification and Authentication

Classification
Procedure
Definition
Reads user identification and authentication information using a buff_io object and determines if the data formulates a valid session level request. Returns a user_ia_struct for use by the session server in finding the proper protocol server is the request is valid.
Responsibilities
Ensure only valid requests get declared valid. Interface with the STOP OS to utilize the existing security databases.
Constraints
MAX_USER_NAME is the maximum length of the user name (20) including the delimiter (\n). MAX_USER_PWD is the maximum length of the user password (10) including the delimiter (\n). MAX_IL_LEN is the maximum length of the integrity level input string (4) including the delimiter (\n). MAX_SL_LEN is the maximum length of the security level input string (4) including the delimiter (\n).
Composition
Refer to "user_ia.h" in Appendix C.
Uses/Interactions
Refer to "shm.h" in Appendix C for the #includes. Uses a valid TCP/IP socket and a buff_io object in addition to STOP OS security database calls to be determined.

Resources

`// Purpose: Pass user IA information primarily when determining`
`//  user's desired session level and validity of login request`
`struct user_ia_struct {`
`  int valid;`
`  char uname [MAX_USER_NAME];`
`  int sl;`
`  int il;`
`};`

Processing
Refer to "user_ia.c" in Appendix C.
Interface/Exports

Refer to "user_ia.h" in Appendix C for a full description of each function.
`struct user_ia_struct user_IA(int sockfd,`
`  struct in_buff_struct *queue );`

6.10 TPS Utilities: Check Secure Attention Signal

Classification
Procedure
Definition
Provided verification of the Secure Attention Signal and extraction of the hardware ID.
Responsibilities
Ensure only valid SASs is accepted. A valid SAS is one that is formatted properly, signed with a valid public-key and contains the hardware ID associated with the public-key in the Connection Database. Return hardware ID from a valid SAS.

Constraints

```
#define MAXHWID             7     // maxsize of hw_id in char + 3 =>
                                  //    hw_id can be 3 digits
#define TELNET_SEND       255   // value for brk
#define TELNET_BRK        243   // value for send
#define MIN_SAK_LEN         3     // minimum valid SAK length
```

SAS format is:

"TELNET_SEND TELNET_BRK <hardware ID><new line(ASCII 10)>"

This will change as encryption is added. The <hardware ID> is 1-3 digits long.

Composition

Refer to "tps_util.h" in Appendix C.

Uses/Interactions

Refer to "tps_util.h" in Appendix C for the #includes.

Uses a valid TCP/IP socket and a buff_io object. buff_io:get_token() is used to extract a delimited char sequence of limited length.

Resources

None.

Processing

Refer to "tps_util.c" in Appendix C.

Interface/Exports

Refer to "tps_util.h" in Appendix C for a full description of each function.

```
int check_SAK(int sockfd, int * hw_id,
              struct in_buff_struct *queue );
```

6.11 TPS Utilities: Socket Relay

Classification

Procedure

Definition
Relay information, possibly between two distinct security levels, from a TCP/IP socket and a pseudo-socket.
Responsibilities
Ensure only information from a security level greater than (sl0, il3) is only routed to a trusted destination and is encrypted (if required) prior to data transfer. Efficiently transfer data with out busy waiting and/or unnecessary blocking. Minimize time using privileges.
Constraints
Privileges are as assigned in priv_util.c, see Appendix C.
Composition
Refer to "tps_util.h" in Appendix C.
Uses/Interactions
Refer to "tps_util.h" in Appendix C for the #includes. Uses a valid TCP/IP socket, a user_ia_struct, a shared memory structure, and a buff_io object.
Resources
None.
Processing
Refer to "tps_util.c" in Appendix C.
Interface/Exports

Refer to "tps_util.h" in Appendix C for a full description of each function.
`int socket_relay(int cli_fd,`
`struct in_buff_struct *cli_buff,`
`struct user_ia_struct *ia_data );`

6.12 Trusted Path Server (TPS)

Classification
Procedure
Definition
Accepts connection requests and creates a child Session Server via the *fork()* function to service the request.
Responsibilities
Properly setup and bind to a reserved port to service connection requests. Act as the driver for all connection requests.
Constraints
#define SERV_PORT 6002 // port TPS will listen to.
Composition
Refer to "tps.c" in Appendix C.
Uses/Interactions
Refer to "tps.c" in Appendix C for the #includes. Uses a valid TCP/IP socket and a buff_io object.
Resources
None.
Processing
Refer to "tps.c" in Appendix C.
Interface/Exports
main()

6.13 Pseudo-Socket Interface

Classification
Wrapper
Definition
Provides socket-like interface for a protocol server to the shared memory structure.

Responsibilities
Mimic the behavior of the socket calls defined in Reference 29.
Constraints
#define I_NREAD 1 // this means ioctl needed #define AF_INET -1 // only socket type supported internet stream. #define SOCK_STREAM -1
Composition
Refer to "pskt.c" in Appendix C.
Uses/Interactions
Refer to "pskt.c" in Appendix C for the #includes and Reference 29 for exact behavior to be mimicked.
Resources

`// used to mimic select timeval parameter.`
`struct timeval {`
`  int tv_sec;`
`  int tv_usec;`
`};`

Processing
Refer to "pskt.c" in Appendix C.

Interface/Exports

Refer to "pskt.h" in Appendix C for a full description of each function.

```
int socket(int domain, int type, int protocol );
int bind(int sockfd, const struct sockaddr * serv_addr, int size );
int listen( int fd, int queue_size );
int accept(int listen_sem, struct sockaddr * addr,
           int * addr_len );
int my_read(int fd, char *buff, int read_limit);
void my_close( int fd );
int my_write(int fd, const char* data, int nbytes );
int select( int bits_to_check, fd_set  *ibits, fd_set *obits,
            fd_set *xbits, struct timeval *timeout );
void FD_SET(int fd, fd_set *bits);
void FD_ZERO(fd_set *bits);
int  FD_ISSET(int fd, fd_set *bits);
void FD_CLEAR(int fd, fd_set *bits);
```

APPENDIX C. SECURE LAN SERVER SOURCE CODE

Makefile for Trusted Path Server

```
source = priv_util.c tps.c util.c tps_util.c cdb.c io_util.c cdb.c
buff_io.c shm.c msem.c listenq.c shm_struct.c user_ia.c
headers = tps_util.h buff_io.h
CFLAGS = -DOSS_OPTION
CF_POLL = -DOSS_OPTION -DUSE_POLL

oss: ${source}
   cc ${CFLAGS} -oss -I/usr/include/sys/ ${source} -o tps -lsocket

osspoll: ${source}
   cc ${CF_POLL} -oss -I/usr/include/sys/ ${source} -o tps -
lsocket

tps_util.o: tps_util.c tps_util.h util.h buff_io.h

app: ${source}
   cc -DUSE_POLL -I/usr/include/sys/ ${source} -o tps -lsocket -
lcass

clean:
   /bin/rm -f /usr2/sdheller/wip/*.o
   /bin/rm -f /usr2/sdheller/wip/core

rm:
   /bin/rm -f /usr2/sdheller/wip/*.o
   /bin/rm -f /usr2/sdheller/wip/core
   /bin/rm -f /usr2/sdheller/wip/tcbe/*.o
   /bin/rm -f /usr2/sdheller/wip/tcbe/core
   /bin/rm -f /usr2/sdheller/wip/echo/*.o
   /bin/rm -f /usr2/sdheller/wip/echo/core

arch:
   tar -cvf `date +../archive/wip_%y%m%d_%H%M.tar`
/usr2/sdheller/wip/*.*

depend:
   cc -Hmake -oss -I/usr/include/sys/ ${source}  -o tps -lsocket -
lcass
```

```
 1 // File: buff_io.h
 2 // Author: Scott Heller
 3 // Date:   2 Feb 1999
 4 // Purpose: buffered IO from network. Uses circular queue data
   structure.
 5 //  will not overwrite data in queue.
 6
 7
 8 #ifndef BUFF_IO_H_
 9 #define BUFF_IO_H_
10
11 #include <errno.h>
12 #include <memory.h>
13
14
15 #ifndef USE_P_SOCKET
16 #include <sys/time.h>
17 #include <sys/types.h>
18 #include <sys/select.h>
19
20 #include <stropts.h>
21 #include <sys/socket.h>
22 #include <unistd.h>
23 #endif //!USE_P_SOCKET
24
25 //for poll
26 #ifdef USE_POLL
27 #include    <stropts.h>
28 #ifdef OSS_OPTION
29 #include    <sys/poll.h>
30 #else  //OSS_OPTION
31 #include    <poll.h>
32 #endif //OSS_OPTION
33 #endif //USE_POLL
34
35
36 #include  "util.h"
37
38
39 #define INBUFFSIZE 4096
40
41 // data struct for circular queue.
42 struct in_buff_struct
43 {
44       char in_buff[INBUFFSIZE];
45       int  read_idx;
46       int  write_idx;
47 };
48
49 #ifdef USE_P_SOCKET
50 #include "echo/pskt.h"
51 #endif //USE_P_SOCKET
52
53
54 // Param:    this: buffer to initialize
55 // Purpose:  initialize buffer index variables.
56 //  does not allocate memory.
```

```
57 void init_buffer( struct in_buff_struct * this );
58
59
60 // Return:  -1 if socket not valid. 1 if data availible.
61 // 0 if no data availible, but socket fd still valid
62 // Param:   fd: valid socket descriptor
63 //          milliseconds: maximum time to delay. actually
64 //    converted to seconds using integer math any value
65 //    less than 1000 will yield a delay of 0.
66 // Purpose: Discover if socket i/o will block or not.
67 int poll_ok_to_read( int fd );
68 int poll_ok_to_read_block( int fd, int milliseconds );
69
70
71 // Param:   fd: valid socket descriptor
72 // Purpose:  Similar to above. Not yet implimented.
73 int poll_ok_to_write( int fd );
74
75
76
77 // Param: fd is a socket descriptor. queue is a buffer.
78 //       delim is the delimiter used to mark the end of the
79 //       desired token. nbytes is the maximum number of bytes
80 //       that will be checked while searching the buffer for
81 //       the delimiter.
82 // Return: char * to the token null terminated with delim removed.
83 //         NULL if delim not found within nbytes.
84 // Notes: The socket need not be open for this call to succeed if
85 //          the data is still in the buffer.
86 char * get_token( int fd, struct in_buff_struct *queue,
87                      const char delim, int nbytes );
88
89
90 // Return:   char * to null terminated string.
91 // Param:    queue: buffer to extract all data from.
92 // Purpose:  return a null terminated string to all remaining data
93 //    in the queue. The queue will be empty after this call.
94 char *empty_buff(struct in_buff_struct *queue);
95
96
97 // Return:  number of char read.
98 // Param:   this: buffer to read from
99 //          data: buffer to place read data in.
100 //          n:    max number of bytes read.
101 // Purpose:  mimic system read.
102 int buff_io_read(struct in_buff_struct * this, char * data, int
    n);
103
104
105 // Return:  number of data bytes in queue.
106 // Param:   queue: queue to query.
107 // Purpose:  return the number of char in the queue. Does not
    include
108 // space for the null terminator if you wish to allocate
109 //  memory for a call to empty_buff() allocate num_char + 1.
110 int num_char(struct in_buff_struct *queue );
111
```

```
112
113 // Return:  next char in queue
114 // Param:   queue: queue to query.
115 // Purpose: return the next char in the queue. No side effects.
116 char peek_char( struct in_buff_struct *queue);
117
118
119 // Return:  next char in queue
120 // Param:   queue: queue to query.
121 // Purpose: return the next char in the queue. char is removed
    from queue.
122 char remove_char(struct in_buff_struct *queue);
123
124
125 // Return:  number of bytes free in the queue
126 // Param:   queue: queue to query.
127 // Purpose: return the number of bytes free in the queue.
128 int bytes_free( struct in_buff_struct * queue);
129
130
131 // Return:  true if empty, false otherwise.
132 // Param:   queue: queue to query.
133 // Purpose:  return true if the queue is empty.
134 int empty( struct in_buff_struct *queue );
135
136
137 // Param:   data: data to add to the buffer
138 //          queue:buffer to add data to
139 //          num_read: number of bytes to add.
140 // Purpose:  add the char data in data[] to the queue.
141 // WARNING: if num_read > bytes free in queue a fatal error is
    generated.
142 //    the program will exit(-1).
143 void add_data( const char *data, struct in_buff_struct *queue, int
    num_read );
144
145
146 // Return:  number of bytes actually added. -1 on error.
147 // Param:   data: data to add
148 //          queue: buffer to add data to
149 //          num_read: number of bytes to add
150 // Purpose:  same as above, but may write less than num_read.
151 //    returns the number of bytes added.
152 int add_data_part( const char *data, struct in_buff_struct *queue,
    int num_read );
153
154
155 // Pre:
156 // Post:
157 // Return:  number of bytes read or -1 on error.
158 // Param:   fd: valid socket descriptor
159 //          queue: desitnation for the newly read data.
160 // Purpose: read upto the number of bytes free in the queue
161 //    return the result of the read() call.
162 int get_data(int fd, struct in_buff_struct *queue );
163
164
```

```
165 // Param:     queue: queue to print
166 // Purpose:  used for debugging, prints contents of the queue.
167 void print_buff_queue( struct in_buff_struct *this );
168
169 #endif
```

```
 1 // File:    cdb.h
 2 // Author:  Scott D. Heller & Susan Bryer-Joyner
 3 // Date:    28 January 1999
 4 // Purpose: Provide interface for manipulating the Connection
   Database(CDB)
 5
 6
 7 #ifndef CDB_H_
 8 #define CDB_H_
 9
10 #include <stdlib.h>
11 #include <string.h>
12 #include <sys/types.h>
13
14 #include "util.h"
15
16 #define MAX_CLIENT      10 // maximum number of CDB records
17 #define MAX_RECORD_LEN  20 // max size of a CDB record
18
19 struct cdb_record
20 {
21    unsigned hw_id;
22    unsigned public_key;
23    unsigned capid;
24 };
25
26
27 // creates a copy of the connection database in RAM
28 // Return: the number of records, exits on error
29 int init_cdb( );
30
31
32 // update the CDB record containing hw_id with new_CAP
33 // num_records must contain the number of records in the CDB
34 // Otherwise it is possible to read from memory not assigned.
35 // Return: true if record found, false if record not found.
36 int update_CDB( int hw_id, int new_CAP );
37
38 // return capid if hw_id in CDB else return -1.
39 int get_CAPID( int hw_id );
40
41
42 // prints one CDB record in RAM.
43 void print_cdb_record(struct cdb_record * this_record );
44
45
46 // print the contents of the CDB in RAM from cdb_ptr for
   num_records.
47 void print_cdb();
48
49 #endif
```

```
 1 // File:    io_util.h
 2 // Author:  Scott D. Heller & Susan Bryer-Joyner
 3 // Date:    1 Feb 1999
 4 // Purpose: IO routines that do not require any trusted includes.
 5
 6
 7 #ifndef IO_UTIL_H_
 8 #define IO_UTIL_H_
 9
10 #include <unistd.h>
11 #include <errno.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <memory.h> // for memset()
15
16 #ifdef USE_P_SOCKET
17 #include "echo/pskt.h"
18 #endif
19
20 #include "util.h"
21
22
23 // Pre:
24 // Post:
25 // Return:  -1 if socket is closed; 0 otherwise
26 // Param:   fd: valid socket descriptor
27 //          ptr: ptr to the data to write
28 //          nbytes: number of bytes to write.
29 // Purpose:  writes n bytes to a descriptor from the buffer ptr
30 int Writen(int fd, void *ptr, size_t nbytes);
31
32
33 // Param:    buff: char string.
34 // Purpose:  debugging. Prints a char string 1 char at a time
35 void print_buff( char *buff );
36
37
38 #endif
```

```
 1 // File: listenq.h
 2 // Author: Scott Heller
 3 // Date:   20 February 1999
 4 // Purpose: Provide a queue of information required to establish a
 5 //  shared memory connection between two processes.
 6
 7 #ifndef LISTENQ_H_
 8 #define LISTENQ_H_
 9
10 #include <stdio.h>
11
12 #include "msem.h" // for lq_init() and lq_remove()
13 #include "priv_util.h" // for calc_key()
14
15 #define MAX_LQ_SIZE   10       // size of the listen queue
16 #define LISTEN_Q_SEM_KEY    5000
17
18 #define MAX_SHM_CONNECTIONS MAX_LQ_SIZE // fix this.
19
20
21 struct listen_q_struct {
22    int                  initialized;  //according to ANSI C static
   var
23                                        // init to zero.
24    int                  write_idx;
25    int                  read_idx;
26    int                  in_buff[MAX_LQ_SIZE]; // key to open shm
   segment.
27    int                  listen_q_sem;
28    key_t                listen_q_sem_key;
29 };
30
31
32 // Pre:     Memory for this is allocated.
33 // Param:   this: ptr to listen_q_struct
34 // Purpose: Initialize buffer index variables and create
35 //  lq semaphore so that accept can block.
36 void lq_init(struct listen_q_struct * this );
37
38
39 // Param:   this: ptr to listen_q_struct to be printed.
40 // Purpose: debugging feed back. prints all data in the
41 //    listen_q_struct *this is decimal and char format.
42 void lq_print(struct listen_q_struct * this );
43
44
45 // Return:  1 if added, 0 if queue full and item not added
46 // Param:   data: integer to be added to listen queue
47 //          queue: where to add the data.
48 // Purpose:  add data from char[] to queue
49 int lq_add_item(int data, struct listen_q_struct *queue );
50
51
52 // Return:  true if empty false otherwise
53 // Param:   queue: queue to test.
54 // Purpose: test if queue is empty.
55 int lq_empty(struct listen_q_struct * queue );
```

```
56
57
58 // Return:  number of free bytes in queue
59 // Param:   queue: queue to query.
60 // Purpose: return number of free bytes in queue
61 int lq_num_free(struct listen_q_struct * queue );
62
63
64 // Return:  integer data stored in queue.
65 // Param:   queue: where to get the data.
66 // Purpose: remove and return one struct relay_struct from queue
67 //    will block until data is availible.
68 int lq_remove_item(struct listen_q_struct * queue );
69
70
71 // Return:  int value of head of queue
72 // Param:   queue: where to get the data.
73 // Purpose: return value of head of queue without removing.
74 int lq_peek_item(struct listen_q_struct *queue );
75
76
77 // Return:  int number of items in queue
78 // Param:   queue: queue to query.
79 // Purpose: return current size of queue
80 int lq_num_items(struct listen_q_struct *queue);
81
82
83 #endif
84
```

```
 1 // File: msem.h
 2 // Author: Richard Stevens
 3 // Modified: Scott Heller: added ring 2 compatibility
 4 //    and the capability do declare an array of sems.
 5 // Date:    27 Feb 99
 6
 7
 8 #ifndef MSEM_H_
 9 #define MSEM_H_
10
11 #ifdef OSS_OPTION
12 #include <stdtyp.h>
13 #include <semaphore.h>
14 #include <error_code.h>
15 #endif // OSS_OPTION
16
17 #include <sys/types.h>
18 #include <sys/ipc.h>
19 #include <sys/sem.h>
20 #include <stdio.h>
21
22 #include <errno.h>
23 #include "util.h"
24
25 #ifndef OSS_OPTION
26 extern int  errno;
27 #endif // OSS_OPTION
28
29 // Return:  semaphore system identifier.
30 // Param:   key: Key of sem to create or open
31 //          initial: initial value of each sem in the array
32 //          num_sems: number of sems in the array
33 // Purpose: create array of sems with initial value or open
34 int sem_create( key_t key, int initial, int num_sems );
35
36
37 // Return:  semaphore system identifier.
38 // Param:   key: Key of sem to open.
39 // Purpose:  open (must already exist)
40 int sem_open ( key_t key);
41
42
43 // Param:    semid: semaphore identifier
44 //           idx:   index of semaphore in the set.
45 // Purpose:  wait = P = down by 1
46 void sem_wait(int id, int idx);
47
48
49 // Param:    semid: semaphore identifier
50 //           idx:   index of semaphore in the set.
51 // Purpose:  signal = V = up by 1
52 void sem_signal(int id, int idx);
53
54
55 // Param:    semid: semaphore identifier
56 //           idx:   index of semaphore in the set.
57 //           value: amount by which to modify semaphore.
```

```
58 //              must not be zero.
59 // Purpose:  General semaphore operation.
60 //    wait if (amount < 0) or signal if (amount > 0)
61 void sem_op(int id, int idx, int value);
62
63
64 // Param:    semid: semaphore identifier
65 // Purpose: close the semaphore set.
66 void sem_close(int);
67
68
69 // Param:    semid: semaphore identifier
70 // Purpose: remove (delete)
71 void sem_rm(int);
72
73 #endif
74
```

```
 1 // File:    priv_util.h
 2 // Author:  Scott Heller
 3 // Date:    27 Feb 99
 4 // Purpose: One stop shopping for gaining the privileges needed to
 5 //  execute as a Secure Session Server. App needs to first be
   installed
 6 //  with privileges using "tp_edit" with administrator access.
 7
 8 #ifndef  PRIV_UTIL_H_
 9 #define  PRIV_UTIL_H_
10
11
12 #ifdef OSS_OPTION
13 #include <stdtyp.h>
14 #include <error_code.h>
15 #include <procman.h>
16 #include <tcb_gates.h>
17 #include <access.h>
18 #include <privileges.h>
19
20 #else // ring 3 application
21
22 #include <sys/types.h>
23 #include <level.h>      // need -lcass to be linked.
24
25 extern int getlevel(const char path[], access_ma *buf );
26
27 #endif //OSS_OPTION
28
29 #include "util.h"
30
31 struct level_struct {
32    int il;
33    int sl;
34 };
35
36 // Return:  The previous privilege set.
37 // Purpose: Enable the fixed set of privileges needed to freely
   communicate
38 //    between two MAC levels.
39 ushort   enable_priv();
40
41
42 // Param:   priv: privilege set.
43 // Purpose: Enable the privilege set defined by priv. Used to
   restore
44 //    privileges after an enable_priv() call.
45 void     set_priv( ushort priv );
46
47
48 // Return:  -1 on error. 0 on success.
49 // Param:   lvl: a level_struct.
50 // Purpose: Get the current integrity and security level of the
   calling
51 //    process. Works in ring 2 and 3.
52 int      get_current_level( struct level_struct * lvl );
53
```

```
54
55 // Return:  sl * 10 + il + base
56 // Param:   base: Shared mem or semaphore base key. Must be
57 //             a multiple of 100 to fuction properly with
58 //             SSS.
59 // Purpose: Given a base key calculate a unique key for
60 //    the current (sl,il) pair.
61 key_t    calc_key( int base );
62
63 #endif //PRIV_UTIL_H_
```

```
 1 // File:     shm.h
 2 // Author:   Scott Heller
 3 // Date:     17 Feb 99
 4 // Purpose: Create and manipulate a shared memory segment
 5 //    between ring 2 and 3.
 6
 7 #ifndef  U_SHM_H_
 8 #define  U_SHM_H_
 9
10 #include    <sys/ipc.h>
11 #include    <sys/shm.h>
12 #include    "util.h"
13 #define     SHM_PERM 00666
14
15 // Return:  Shared memory identifier
16 // Param:   key: The key to find the shared mem segment.
17 //          addr: Will be set to the first address of the segment
18 //          size: size of the desired segment.
19 // Purpose: create and attach to a shared memory segment
20 //    make this segment availible to the protocol server
21 int get_shm(key_t key, void ** addr, size_t size );
22
23 // Param:   shmid: the shared mem segment identifier
24 // Purpose: return a ptr to a shm segment from a shmid
25 void *attach_shm( int shmid );
26
27 // Param:   shm_id: shared memory identifier
28 //          addr:   address of shared memory.
29 // Purpose: detach and remove shared memory segment from the
   system
30 //    this must be done every time prior to exit being called
31 void remove_shm( int shm_id, void * addr );
32
33
34 #endif // SHM_H_
35
```

```
 1 // File: shm_struct.h
 2 // Author: Scott Heller
 3 // Date:  22 Feb 1999
 4 // Purpose: Provide interface for connection shm. One shm segment
 5 //    per level.
 6
 7 #ifndef SHM_STRUCT_H_
 8 #define SHM_STRUCT_H_
 9
10 #ifdef OSS_OPTION
11 #include <tcb_gates.h>
12 #include <shared_mem.h>
13 #endif // OSS_OPTION
14
15 #include "util.h"
16 #include "buff_io.h"
17 #include "io_util.h"
18 #include "user_ia.h"
19 #include "listenq.h"
20 #include "shm.h"
21 #include "msem.h"
22 #include "priv_util.h"
23
24 #define MAX_OPEN_CONN 5
25
26 // The following keys may be changed, but
27 // must be multiples of 100 inorder for the SSS
28 // to find the proper level based upon the keys.
29 // Acutal keys used to open shm and sems are:
30 //  key = sl * 10 + il + base;
31 #define SHM_STRUCT_BASE_KEY 7800
32 #define LEVEL_SEM_KEY_BASE  8000
33
34 #ifdef USE_P_SOCKET
35 // Needed to fully impliment pseudo sockets.
36 struct sockaddr {
37    char sa_len;
38    char sa_family;
39    char   sa_data[14];
40 };
41
42 #endif //USE_P_SOCKET
43
44 // Each p_socket connection needs an inbound and outbound
45 // buffer as well as a flag to indicate if in_use or not.
46 // The addr should eventually be filled in by the SSS
47 // and returned in accept().
48 struct connect_struct {
49     int in_use;
50     struct in_buff_struct to_svr_buff;
51     struct in_buff_struct to_cli_buff;
52     struct sockaddr addr; // client address storage
53 };
54
55 // The entire contents of each level's shared memory segment
56 // lq is used to block on by accept.
57 struct shm_hdr {
```

```
58      struct listen_q_struct lq;
59      struct connect_struct conn[MAX_OPEN_CONN];
60      int     shm_hdr_shmid; // needed for ss_cleanup call to rm shm.
61      int     conn_semid;
62  };
63
64  // Return:  -1 on error. shmid on success
65  // Param:   shm_ptr: pass the address of the pointer
66  //  to a shm_hdr struct. It will be set to the first
67  //  address in the new shared memory segment.
68  // Purpose: Allocate and intialize shared memory
69  //    for the current level of the protocol server.
70  int init_shm_hdr( struct shm_hdr ** shm_ptr );
71
72
73  // Param:   shm_ptr: first address in the shared mem segment.
74  // Purpose: Remove shared memory from the system.
75  void ss_cleanup( struct shm_hdr * shm_ptr );
76
77  // Return:  shared memory id. -1 on error.
78  // Purpose: Don't create, just get the shm struct address
79  //    for the (sl,il) pair in ia_data.
80  int ss_get_hdr(struct shm_hdr ** shm_hdr,
81                 struct user_ia_struct * ia_data);
82
83  // Param:   shm_hdr: pointer to the shared mem segment.
84  // Purpose: detach shm seg from current process
85  void ss_detach_hdr( struct shm_hdr * shm_hdr );
86
87  // Return:  number of char read or writen
88  // Param:   fd: index into the connection array
89  //          shm_hdr: ptr to shared mem segment
90  //          others as expected for read and write.
91  // Purpose: I/O functions for pseudo socket calls
92  int ss_read(int fd, struct shm_hdr * shm_hdr,
93              char *buff, int nbytes );
94  int ss_write(int fd, struct shm_hdr *shm_hdr,
95               const char *data, int nbytes );
96
97
98  // Return:  number of char read or writen
99  // Param:   fd: index into the connection array
100 //          shm_hdr: ptr to shared mem segment
101 //          others as expected for read and write.
102 // Purpose: I/O functions for SSS calls to shm_struct
103 int ss_read_fm_svr(int fd, struct shm_hdr * shm_hdr,
104                    char *buff, int nbytes );
105 int ss_write_to_svr(int fd, struct shm_hdr *shm_hdr,
106                     const char *data, int nbytes);
107
108 // Param:   fd: index into the connection array
109 //          shm_hdr: ptr to shared mem segment
110 // Purpose: mark the connection fd as not in use.
111 void ss_close(int fd, struct shm_hdr *shm_hdr );
112
113
114 // Return:  boolean indicating results of test.
```

```
115 // Param:   idx: index into the connection array
116 //          shm_hdr: ptr to shared mem segment
117 // Purpose: tests needed by select() in pskt.h
118 int ss_data_avail(  int idx, struct shm_hdr *shmhdr );
119 int ss_space_avail( int idx, struct shm_hdr *shmhdr );
120 int ss_socket_error(int idx, struct shm_hdr *shmhdr );
121
122
123 // Return:  index of connection from listen queue.
124 // Param:   shm_hdr: ptr to shared mem segment
125 // Purpose: used by accept to block until connection is
126 //    availible.
127 int ss_block_on_lq( struct shm_hdr *shmhdr );
128
129
130 // Param:   shmhdr: ptr to shared mem segment
131 //          idx: index into the connection array
132 //          from: buffer to copy into shared memory.
133 // Purpose: Used by session_relay to copy the TCBE client buffer
134 //   into shared memory.
135 void ss_copy_cli_buff( struct shm_hdr *shmhdr, int idx,
136                        struct in_buff_struct *from );
137
138
139 // Return:  number of bytes transferd. -1 on error.
140 // Param:   shmhdr: ptr to shared mem segment
141 //          idx: index into the connection array
142 //          sockfd: valid socket descriptor
143 // Purpose: Transfer data between a socket and a shared mem
    buffer.
144 int ss_xfer_skt_buff( struct shm_hdr *shmhdr, int pskfd, int
    sockfd );
145 int ss_xfer_buff_skt( struct shm_hdr *shmhdr, int pskfd, int
    sockfd );
146
147
148 // Pre:
149 // Post:
150 // Return:  New pskt connection index. -1 if no connection free.
151 // Param:   shmhdr: ptr to shared mem segment
152 // Purpose: Find next availible connection in shared memory. Then
153 //    allocate that connection for use by the SSS.
154 int ss_request_connection( struct shm_hdr *shmhdr );
155
156 #endif
157
```

```
 1 // File:    tps_util.h
 2 // Author:  Scott D. Heller & Susan Bryer-Joyner
 3 // Date:    28 January 1999
 4 // Purpose: Functions used by the Trusted Path Server (tps.c)
 5
 6
 7 #ifndef TPS_UTIL_H_
 8 #define TPS_UTIL_H_
 9
10 #include <unistd.h>
11 #include <errno.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <memory.h> // for memset()
15
16 //for select in select_sleep
17 #include <sys/time.h>
18 #include <sys/types.h>
19 #include <sys/select.h>
20
21 #include "cdb.h"
22 #include "io_util.h"
23 #include "buff_io.h"
24 #include "util.h"
25 #include "shm.h"
26 #include "msem.h"
27 #include "listenq.h"
28 #include "shm_struct.h"
29 #include "priv_util.h"
30
31
32 #define MAX_SAK_ATTEMPTS   3     // limit of invalid SAK attempts
   before exit
33 #define MAXHWID            7     // maxsize of hw_id in char + 3
   =>
34                                  //    hw_id can be 3 digits
35 #define TELNET_SEND       255   // value for brk
36 #define TELNET_BRK        243   // value for send
37 #define MIN_SAK_LEN         3    // minumum valid SAK length
38 #define SERV_PORT        6002  // port TPS will listen to.
39
40
41 // Return:  -1 if not valid SAK mssg else return CAPID
42 // Param:   sockfd: valid socket descriptor
43 //          hw_id : pointer used to return the hardware id of the
   TCBE
44 //          queue : buffer associated with the current connection.
45 // Purpose: Verify the SAS is legitiment. Eventually public-key
   verification
46 //    should occur here.
47 // Note:    A valid SAK mssg, for now, is one the starts:
48 //  "send brk" and is followed by a 1-3 digit hardware ID.
49 int check_SAK(int sockfd, int * hw_id, struct in_buff_struct
   *queue );
50
51
52 // Return:  -1 on error.
```

```
53 // Param:   cli_fd: valid socket descriptor for communicating with
   TCBE
54 //          cli_buff: buffer used to store data from TCBE
55 //          ia_data:  sl and il for desired current session.
56 // Purpose: relay data from TCBE to protocol server and vice a
   versa.
57 //    This is were most of the real work of the Secure Session
   Server
58 //    is accomplished.
59 int socket_relay(int cli_fd,
60                  struct in_buff_struct *cli_buff,
61                  struct user_ia_struct *ia_data    );
62
63
64 // Param:   sockfd: valid socket descriptor
65 //          seconds: maximum number of seconds to sleep.
66 // Purpose: Select test. Any activity on sockfd will cause
67 //    immediate return.
68 void select_sleep(int sockfd, long seconds );
69
70
71 #endif
```

```
 1 // File:    user_ia.h
 2 // Author:  Scott D. Heller & Susan Bryer-Joyner
 3 // Date:    28 January 1999
 4 // Purpose: Perform User Identification and Authentication.
 5
 6
 7 #ifndef USER_IA_H_
 8 #define USER_IA_H_
 9
10 #include <unistd.h>
11 #include <errno.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14
15 #include "io_util.h"
16 #include "buff_io.h"
17 #include "util.h"
18
19 #define MAX_USER_NAME    20    // maximum length of user name
20 #define MAX_USER_PWD    10
21 #define MAX_IL_LEN         4
22 #define MAX_SL_LEN         4
23
24
25 // Purpose: Pass user IA information primarily when determining
26 //  user's desired session level and validity of login request
27 struct user_ia_struct {
28  int valid;
29  char uname [MAX_USER_NAME];
30  int sl;
31  int il;
32 };
33
34
35 // Pre:     sockfd is connected to a valid socket and a SAS was
   recieved.
36 // Post:    user IA data will be consumed from the (sockfd,queue).
37 // Return:  user_ia_struct
38 // Param:   sockfd from where to get the user information.
39 //          queue currently being used with sockfd to store
   inbound data.
40 // Purpose: Perform user idenfication and authentication
41 //    return true if valid.
42 struct user_ia_struct user_IA(int sockfd, struct in_buff_struct
   *queue );
43
44
45 #endif
```

```
 1 // File:    util.h
 2 // Author:  Scott D. Heller & Susan Bryer-Joyner
 3 // Date:    28 January 1999
 4 // Purpose: General utility functions possibly used by any
   application
 5
 6 #ifndef UTIL_H_
 7 #define UTIL_H_
 8
 9 #include <unistd.h>
10 #include <errno.h>
11 #include <stdio.h>
12 #include <stdlib.h>
13
14
15 extern int errno;
16
17 #ifndef true
18 #define true 1
19 #endif
20
21 #ifndef false
22 #define false 0
23 #endif
24
25 // Param:   test value
26 // Purpose: if test == 0 ensure will print perror information
27 //    and exit.ring 2 application do not have access to assert.
28 void ensure(int test);
29 void ensure_m(int test, char *mssg);
30
31
32 // Param:   int on/off switch, debugging message.
33 // Purpose: Standardized debugging. If int is not zero print the
34 //    string prefaced by the pid of the calling process
35 void dbug( int on, char * prompt );
36
37
38 // Param:   on/off swtich, debugging prompt, integer data
39 // Purpose: Standardized debugging.  If int is not zero print the
40 //    string followed by value of int prefaced by the pid of the
41 //    calling process.
42 void dbugd( int on, char *prompt, int data);
43
44
45 #endif
46
```

```
 1 // File: buff_io.c
 2 // Author: Scott Heller
 3 // Date:   2 Feb 1999
 4 // Purpose: Buffered IO from/to network
 5
 6 #include "buff_io.h"
 7
 8
 9 // initialize buffer index variables.
10 //  does not allocate memory.
11 void init_buffer(struct in_buff_struct * this )
12 {
13       int debug_on = 0;
14
15       this->write_idx = 0;
16       this->read_idx = 0;
17
18       if(debug_on) print_buff_queue(this);
19
20 }// end init_buffer
21
22
23 // Purpose: debugging feed back. prints all data in the
24 //    in_buff_struct *this is decimal and char format.
25 void print_buff_queue(struct in_buff_struct * this )
26 {
27
28       int idx = this->read_idx;
29
30       printf("buffer queue: size = %d: contents:", num_char(this)
   );
31
32       while(idx != this->write_idx )
33       {
34          printf("(%d, %c)", this->in_buff[idx], this->in_buff[idx]
   );
35          idx = (idx + 1)%INBUFFSIZE;
36       }
37       printf("\n");
38 }// end print_buff_queue
39
40 // read upto the number of bytes free in the queue
41 // return the result of the read() call.
42 //    this result should be the number of bytes read
43 //    or -1 on error.
44 int get_data(int fd, struct in_buff_struct *queue )
45 {
46       int debug_on = 0;
47       dbug(debug_on, "get_data: entered.");
48
49       // only read upto the # bytes free in queue
50       int read_limit = bytes_free(queue );
51
52       // allocate memory for incomming data.
53       char *temp_buff = malloc(read_limit);
54       int num_read = 0;
55
```

```
56       // initialize the input buffer.
57       if(!memset(temp_buff, 0, read_limit ) )
58       {
59          perror("buff_io:get_data:memset");
60          exit(-1);
61       }
62
63       // attempt to read until valid error or valid read.
64       do {
65          dbug(debug_on, "buff_io:get_data: ABOUT to read.");
66 #ifdef USE_P_SOCKET
67          num_read = my_read(fd, temp_buff, read_limit );
68 #else //USE TCP/IP Sockets
69          num_read = read(fd, temp_buff, read_limit );
70 #endif // USE_P_SOCKET
71       } while(num_read < 0 && errno == EINTR );
72
73       // if there was an error while reading.
74       if(num_read < 0 )
75       {
76          perror("buff_io:get_data:read");
77       } else if(num_read > 0 ) {
78
79          // move data from temp_buff to queue.
80          add_data(temp_buff, queue, num_read );
81       } // end if
82
83       dbug(debug_on, "buff_io:leaving get_data");
84       if(debug_on) print_buff_queue(queue);
85
86       free(temp_buff );
87
88       return num_read;
89
90 }// end get_data
91
92
93 // add data from char[] to queue
94 // assert: bytes <= bytes_free
95 void add_data(const char *data, struct in_buff_struct *queue, int
   bytes )
96 {
97       int debug_on = 0;
98       dbugd( debug_on, "buff_io:add_data adding nbytes = ", bytes
   );
99       ensure(bytes <= bytes_free(queue) );
100
101      for(int idx = 0; idx < bytes; idx++ )
102      {
103         // add char at write_idx
104         queue->in_buff[ queue->write_idx ] = data[idx];
105
106         // incremetn write_idx unless it would be too big
107         queue->write_idx = (queue->write_idx + 1)%INBUFFSIZE;
108
109      }// end for loop
110
```

```
111       if(debug_on) print_buff_queue( queue );
112
113 } // end add_data
114
115 int add_data_part(const char *data, struct in_buff_struct *queue,
    int bytes )
116 {
117       int free = bytes_free(queue);
118       int nwritten = (bytes > free) ? free : bytes;
119
120       add_data(data, queue, nwritten );
121
122       return nwritten;
123
124 }// end add_data_part
125
126 int empty(struct in_buff_struct * queue )
127 {
128       // true if empty
129       return (queue->read_idx == queue->write_idx );
130 }
131
132 // return number of free bytes in queue
133 int bytes_free(struct in_buff_struct * queue )
134 {
135
136       int result = 0;
137       if(queue->write_idx < queue->read_idx )
138       {
139          result = queue->read_idx - queue->write_idx;
140
141       } else {
142          result = queue->read_idx - queue->write_idx + INBUFFSIZE;
143       }
144
145       return result;
146
147 } // end bytes_free
148
149
150 // remove and return one char from queue
151 char remove_char(struct in_buff_struct * queue )
152 {
153
154       char result = queue->in_buff[ queue->read_idx ];
155
156       queue->read_idx = (queue->read_idx + 1)%INBUFFSIZE;
157
158       return result;
159
160 }// end remove_char
161
162
163 // peek at next char in queue
164 char peek_char(struct in_buff_struct *queue )
165 {
166       return(queue->in_buff[queue->read_idx]);
```

```
167
168 }// end peek_char
169
170 // return true if delim found within nbytes of head of queue
171 int delim_exists(struct in_buff_struct *queue, char delim, int
    nbytes )
172 {
173       int limit = nbytes <= num_char(queue) ? nbytes :
    num_char(queue);
174       int result = false;
175       int curr_idx = queue->read_idx;
176
177       while(limit-- && !result )
178       {
179
180          if(queue->in_buff[ curr_idx ] == delim )
181          {
182             result = true;
183
184          } else {
185             curr_idx = (curr_idx + 1)%INBUFFSIZE;
186
187          }//end if
188
189       } //end while
190
191       return result;
192
193 }
194
195
196 // return current size of queue
197 int num_char(struct in_buff_struct *queue)
198 {
199       return(INBUFFSIZE - bytes_free(queue) );
200 }// end num_char
201
202 // return char * to all reamaining data in queue
203 // queue will be empty afterwards
204 char *empty_buff(struct in_buff_struct *queue)
205 {
206       char *result = malloc(num_char(queue) + 1 );
207       char *curr_ptr = result;
208
209       while(num_char(queue ) )
210       {
211          *curr_ptr = remove_char(queue);
212          curr_ptr++;
213       }
214       // fdd null char to terminate string.
215       *curr_ptr = 0;
216
217       return result;
218
219 }// end empty_buff
220
```

```
221 int buff_io_read(struct in_buff_struct *queue, char *buff, int
    nbytes)
222 {
223       int result = 0;
224
225       for(int idx = 0; idx < nbytes &&  num_char(queue); idx++ )
226       {
227          buff[idx] = remove_char(queue);
228          result++;
229       }
230       return result;
231 }// end buff_io_read()
232
233
234 // return string containing char upto, but not
235 //  including delim, NULL if delim is not in the buffer.
236 char * get_token(int fd, struct in_buff_struct *queue,
237                       const char delim, int nbytes )
238 {
239       int debug_on = 0;
240
241       int ok = 0;
242
243     dbug(debug_on, "buff_io:get_token - entering");
244       char *result = malloc(nbytes);
245       char *curr_ptr = result;
246
247       // if the queue is empty there is no data to read.
248       int done = empty(queue);
249       int read_result = 0;
250       dbug(debug_on, "buff_io:get_token:calling
    poll_ok_to_read_block()" );
251
252       ok = poll_ok_to_read_block(fd, 0 );
253       dbug(debug_on, "biff_io:get_token: poll_ok... returned");
254       dbugd(debug_on, "buff_io:get_token: ok = ", ok );
255       if(ok == 1)
256       {
257          dbug(debug_on,"buff_io:get_token: calling get_data()");
258          // read data avail from stream fd.
259          read_result = get_data(fd, queue);
260          if(debug_on ) print_buff_queue(queue );
261       }
262       if(delim_exists(queue, delim, nbytes ) )
263       {
264          dbug(debug_on, "buff_io:get_token: delim_exists finding
    token " );
265          for(int idx = 0; idx < nbytes && !done; idx++ )
266          {
267             if((*curr_ptr = remove_char(queue) ) == delim )
268             {
269                dbugd(debug_on, "buff_io:get_token: *curr_ptr = ",
    *curr_ptr );
270                *curr_ptr = '\0';
271                done = true;
272             } else {
273                curr_ptr++;
```

```
274
275                 if(read_result < 0 && empty(queue) ) done = true;
276
277              }//end if
278
279           }// end for
280
281           dbug(debug_on, result );
282
283        } else {
284           // the delim is not yet(?) in the queue.
285           free(result );
286           result = NULL;
287
288        }// end if
289
290        dbug(debug_on, "buff_io:get_token - leaving");
291
292        return result;
293
294 }// end get_token
295
296
297 // legacy call to get_data() should be removed.
298 int get_all_avail(int fd, struct in_buff_struct *queue )
299 {
300        return  get_data(fd, queue );
301 }// end get_all_avail
302
303
304 // return -1 if error indicates the socket is closed.
305 // return 0 if data not available or 1 if ok to read.
306 // if calling blocking then milliseconds indicates how
307 //  long the call will wait for a status change of the fd.
308 int poll_ok_to_read(int fd )
309 {
310        return(poll_ok_to_read_block(fd, 0 ) );
311 }// end poll_ok_to_read
312
313 int poll_ok_to_read_block(int fd, int milliseconds )
314 {
315        int debug_on = 0;
316
317        int result = 0;
318
319        fd_set ibits, obits, xbits;
320        FD_ZERO(&ibits); FD_ZERO(&obits); FD_ZERO(&xbits);
321
322        static struct timeval timeout;
323        timeout.tv_sec = milliseconds/1000;
324        timeout.tv_usec = 0;
325
326        FD_SET( fd, &ibits );
327        FD_SET( fd, &xbits );
328
329        select( 16, &ibits, &obits, &xbits, &timeout );
330
```

```
331         if( FD_ISSET( fd, &xbits ) ) result = -1;
332         else if( FD_ISSET(fd, &ibits ) ) result = 1;
333
334         return result;
335
336 }// end poll_ok_to_read
337
338 // return -1 if error indicates socket is closed.
339 // return 1 if ok to write with out blocking,
340 //  0 otherwise.
341 int poll_ok_to_write(int fd )
342 {
343         int debug_on = 0;
344         int result = 0, sigs = 0;
345
346 /*      dbug(debug_on, "buff_io:poll_ok_to_write:entered");
347         if(ioctl(fd, I_GETSIG, &sigs ) < -1 )
348         {
349            result = -1;
350         } else {
351            dbugd(debug_on, "buff_io:poll..write: sigs = ", sigs );
352            result = sigs & S_OUTPUT;
353         }// end if
354 */
355
356 //      return result;
357 return 1;
358
359 }// end poll_ok_to_write
360
```

```
 1 // File:    cdb.c
 2 // Author:  Scott D. Heller & Susan Bryer-Joyner
 3 // Date:    28 January 1999
 4 // Purpose: Provide interface for manipulating the Connection
   Database(CDB)
 5
 6
 7 #include <stdio.h>
 8 #include <stdlib.h>
 9 #include <math.h> // for itostr() which should be moved to util.c
10
11 #include "cdb.h"
12
13 // variable used by all cdb functions.
14 static struct cdb_record CDB_PTR[MAX_CLIENT];
15 static int NUM_RECORDS = 0;
16
17
18 // public functions
19
20
21 // creates a copy of the connection database in RAM
22 // Return: the number of records, -1 on error
23 int init_cdb()
24 {
25
26    int debug_on = 0;
27
28    FILE *db_file;
29
30    char* db_name = "/usr2/sdheller/wip/cdb_file.txt";
31    char record_str[MAX_RECORD_LEN];
32    char *field1;
33    char *field2;
34
35    int num_loaded = 0;
36
37    // assigns file descriptor to open file
38    // if open fails, exits with -1
39    if ((db_file = fopen(db_name, "r")) == NULL)
40    {
41       perror("cdb.c fopen");
42       printf("Problem opening connection database: %s\n",
   db_name);
43       exit(-1);
44    }//end if
45
46
47    // initializes the array with values from the
48    // connection database file
49    while ( num_loaded < MAX_CLIENT && fgets( record_str,
   MAX_RECORD_LEN, db_file ) )
50    {
51       dbug( debug_on, "reading a record");
52       dbug( debug_on, record_str );
53
54       field1 = strtok( record_str, "," );
```

```
55        dbug( debug_on, "field1" );
56        dbug( debug_on, field1 );
57
58        field2 = strtok( NULL, "\n" );
59        dbug( debug_on, "field2" );
60        dbug( debug_on, field2 );
61
62        if( !sscanf( field1, "%d", &(CDB_PTR[num_loaded].hw_id) ))
63        {
64           printf("sscanf failed to convert hw_id from data
   file\n");
65           exit(-1);
66        }
67        dbugd( debug_on, "sscanf( field1 )",
   CDB_PTR[num_loaded].hw_id );
68
69        if( !sscanf( field2, "%d", &(CDB_PTR[num_loaded].public_key)
   ))
70        {
71           printf("sscanf failed to convert public_key from data
   file\n");
72           exit(-1);
73        }
74        dbugd( debug_on, "sscanf( field2 )",
   CDB_PTR[num_loaded].public_key );
75
76        CDB_PTR[num_loaded].capid = 0;
77
78        dbugd(debug_on, "hw_id", CDB_PTR[num_loaded].hw_id );
79        dbugd(debug_on, "pk", (int)CDB_PTR[num_loaded].public_key );
80
81        num_loaded++;
82
83        dbugd(debug_on, "num_loaded: " , num_loaded );
84
85     }//end while
86
87     //close the connection database file
88     fclose( db_file );
89
90       NUM_RECORDS = num_loaded;
91     return num_loaded;
92
93  } // end init_cdb
94
95
96  // update the CDB record containing hw_id with new_CAP
97  // num_records must contain the number of records in the CDB
98  // Otherwise it is possible to read from memory not assigned.
99  // Return: true if record found, false if record not found.
100 int update_CDB( int hw_id, int new_CAP )
101 {
102    int debug_on = 0;
103    int result = false;
104
105    struct cdb_record * curr_ptr = CDB_PTR;
106
```

```
107    // find the hw_id in the database.
108    for(int idx = 0; idx < NUM_RECORDS && hw_id != curr_ptr->hw_id;
    idx++)
109    {
110       curr_ptr++;
111    }
112    // found the hw_id in the database.
113    dbugd(debug_on, "update_CDB: hw_id found in record:", idx);
114    if ( idx < NUM_RECORDS )
115    {
116       result = true;
117       curr_ptr->capid = new_CAP;
118    }
119    if(debug_on) print_cdb_record( curr_ptr );
120    return result;
121
122 }// end update_CDB
123
124
125 // prints one CDB record in RAM.
126 void print_cdb_record(struct cdb_record * this_record )
127 {
128    printf("cdb_record: hw_id %d; public_key %d; capid %d.\n",
129             this_record->hw_id, this_record->public_key,
130             this_record->capid );
131
132 }
133
134
135 // print the contents of the CDB in RAM from cdb_ptr for
    num_records.
136 void print_cdb()
137 {
138    for(int idx = 0; idx < NUM_RECORDS; idx++)
139    {
140       print_cdb_record( &CDB_PTR[idx] );
141    }
142 }// end print_cdb
143
144 // return -1 if hw_id is not found in CDB.
145 // else return the associated CAPID
146 int get_CAPID( int hw_id )
147 {
148       int debug_on = 0;
149
150       int result = -1;
151
152       dbug(debug_on, "entered get_CAPID");
153
154       // find the idx of the record with hw_id.
155       for( int idx = 0; idx < NUM_RECORDS && hw_id !=
    CDB_PTR[idx].hw_id; idx++ );
156
157       // if hw_id was in CDB return the associated capid.
158       if( idx < NUM_RECORDS )
159       {
160          result = CDB_PTR[idx].capid;
```

```
161         }
162
163         dbugd(debug_on, "leaving get_CAPID: capid = ", result );
164
165         return result;
166
167 } // end get_CAPID
168
```

```
 1 // File:    io_util.c
 2 // Author:  Scott D. Heller & Susan Bryer-Joyner
 3 // Date:    1 Feb 1999
 4 // Purpose: IO routines that do not require any trusted includes.
 5
 6 #include "io_util.h"
 7 // function required by Writen( int, const void *, size_t )
 8
 9 // Write "n" bytes to a descriptor.
10 // Author: R. Stevens
11 size_t writen(int fd, const void *vptr, size_t n)
12 {
13      int debug_on = 0;
14
15    size_t      nleft = n;
16    size_t      nwritten = 0;
17    const char  *ptr = vptr;
18
19    while (nleft > 0) {
20
21       dbugd(debug_on, "io_util:writen:attempting write, nleft =",
   nleft);
22       do  {
23 #ifdef USE_P_SOCKET
24             nwritten = my_write(fd, ptr, nleft);
25 #else // use normal tcp/ip sockets
26             nwritten = write(fd, ptr, nleft);
27 #endif // USE_P_SOCKET
28       } while (errno == EINTR && (int)nwritten < 0 );
29
30       if( nwritten > 0 )
31       {
32          nleft -= nwritten;
33          ptr   += nwritten;
34       }
35       else
36       {
37          n = nwritten;
38          break;
39       }//end if
40
41    }// end while()
42
43    return(n);
44 }
45 /* end writen */
46
47
48 // function Writen( fd, void *, size_t )
49
50 // writes n bytes to a descriptor from the buffer ptr
51 // return -1 on error by write().
52 // Author: R. Stevens
53 int Writen(int fd, void *ptr, size_t nbytes)
54 {
55    int result = -1;
56
```

```
57    if ((result = writen(fd, ptr, nbytes)) != nbytes)
58    {
59       //err_sys("writen error");
60       printf("Error: writen error\n");
61    }//end if
62
63    if (result >= 0)
64    {
65       result = 0;
66    }//end if
67
68    return result;
69
70 }// end Writen()
71
72
73 // print a char buffer 1 char at a time
74 void print_buff( char *buff )
75 {
76    char *curr_char = buff;
77
78    while( *curr_char ) printf( "%c", *curr_char++ );
79    printf("\n");
80 }// end print_buff()
81
82
```

```
 1 // File : listenq.c
 2 // Author: Scott Heller
 3 // Date:   20 February 1999
 4 // Purpose: Provide a queue of information required to establish a
 5 //  shared memory connection between two processes.
 6
 7
 8 #include "listenq.h"
 9
10
11 // Initialize buffer index variables and create
12 //  lq semaphore so that accept can block.
13 void lq_init(struct listen_q_struct * this )
14 {
15       int debug_on = 0;
16
17       this->write_idx = 0;
18       this->read_idx = 0;
19
20       // create semaphore for this listen queue. This will need to
   be changed
21       // if more than one listen queue per level is to be used.
22       this->listen_q_sem_key = calc_key( LISTEN_Q_SEM_KEY );
23       this->listen_q_sem = sem_create( this->listen_q_sem_key, 0,
   1 );
24
25       dbugd( debug_on, "lq_init: this->listen_q_sem = ", this-
   >listen_q_sem );
26
27       if(debug_on) lq_print(this);
28
29 }// end  lq_init()
30
31
32 // must call before exit()
33 void lq_remove(struct listen_q_struct * this )
34 {
35    sem_rm( this->listen_q_sem );
36 }// end remove_listen_q
37
38
39
40 // Purpose: debugging feed back. prints all data in the
41 //    listen_q_struct *this is decimal and char format.
42 void lq_print(struct listen_q_struct * this )
43 {
44
45       int idx = this->read_idx;
46
47       printf("listen queue: size = %d: contents:",
   lq_num_items(this) );
48
49       while(idx != this->write_idx )
50       {
51          printf("(%d) ", this->in_buff[idx]);
52          idx = (idx + 1)%MAX_SHM_CONNECTIONS;
53       }
```

```
54        printf("\n");
55  }// end lq_print()
56
57
58  // add data from char[] to queue
59  // Return: 1 if added, 0 if queue full and item not added
60  int lq_add_item(int data, struct listen_q_struct *queue )
61  {
62        int debug_on = 0;
63  #ifdef DEMO
64        debug_on = 1;
65  #endif // DEMO
66        int result = 0;
67        dbugd( debug_on, "lq_add_item: adding data => ", data );
68        if(lq_num_free(queue) >= 1 )
69        {
70           int sem_id = 0;
71           // add relay_struct at write_idx
72           queue->in_buff[ queue->write_idx ] = data;
73
74           // incremetn write_idx unless it would be too big
75           queue->write_idx = (queue->write_idx +
    1)%MAX_SHM_CONNECTIONS;
76           result = 1;
77           sem_id = sem_open( queue->listen_q_sem_key );
78           sem_op( sem_id, 0, 1 ); // incremnt sem - signal
79           sem_close( sem_id );
80
81        }// end if
82
83        return result;
84
85  } // end add_item
86
87  // return: true if empty false otherwise
88  int lq_empty(struct listen_q_struct * queue )
89  {
90        // true if empty
91        return (queue->read_idx == queue->write_idx );
92  }
93
94  // return number of free bytes in queue
95  int lq_num_free(struct listen_q_struct * queue )
96  {
97
98        int result = 0;
99        if(queue->write_idx < queue->read_idx )
100       {
101          result = queue->read_idx - queue->write_idx;
102
103       } else {
104          result = queue->read_idx - queue->write_idx +
    MAX_SHM_CONNECTIONS;
105       }
106
107       return result;
108
```

```
109 } // end num_free
110
111
112 // remove and return one struct relay_struct from queue
113 // will block until data is availible.
114 int lq_remove_item(struct listen_q_struct * queue )
115 {
116       int debug_on = 0;
117       int result = 0, sem_id = 0;
118       dbugd(debug_on, "entered lq_remove_item:sem_key = ",
119          queue->listen_q_sem_key );
120 #ifdef DEMO
121       printf("Blocking on listen queue\n");
122 #endif // DEMO
123
124
125       sem_id = sem_open( queue->listen_q_sem_key );
126       dbugd( debug_on, "lq_remove_item: sem_open: sem_id =",
    sem_id );
127       dbugd( debug_on, "lq_remove_item: queue->listen_q_sem = ",
128          queue->listen_q_sem );
129
130       sem_op( sem_id,0,  -1 );
131       dbug( debug_on, "lq_remove_item:stopped blocking" );
132
133       sem_close(sem_id);
134       dbug(debug_on, "lq_remove_item:closed sem_id");
135       if(debug_on) lq_print(queue);
136       dbug(debug_on, "lq_remove_item taking item off queue");
137       result = queue->in_buff[ queue->read_idx ];
138
139       queue->read_idx = (queue->read_idx + 1)%MAX_SHM_CONNECTIONS;
140
141       return result;
142
143 }// end remove_item
144
145
146 // peek at next struct relay_struct in queue
147 int lq_peek_item(struct listen_q_struct *queue )
148 {
149       return(queue->in_buff[queue->read_idx]);
150
151 }// end peek_item
152
153
154 // return current size of queue
155 int lq_num_items(struct listen_q_struct *queue)
156 {
157       return(MAX_SHM_CONNECTIONS - lq_num_free(queue) );
158 }// end num_items
159
160
  1 /*
  2  * Provide an simpler and easier to understand interface to the
    System V
```

```
 3  * semaphore system calls.  There are 7 routines available to the
    user:
 4  * *
 5  * We create and use a n-member set for the requested semaphore.
 6  * The first member, [0], of the semaphore set is used as a lock
    variable
 7  * to avoid any race conditions in the sem_create() and
    sem_close()
 8  * functions.
 9  * The second member, [1], is a counter used to know when all
    processes
10  * have finished with the semaphore.  The counter is initialized
    to a large
11  * number, decremented on every create or open and incremented on
    every close.
12  * This way we can use the "adjust" feature provided by System V
    so that
13  * any process that exit's without calling sem_close() is
    accounted
14  * for.  It doesn't help us if the last process does this (as we
    have
15  * no way of getting control to remove the semaphore) but it will
16  * work if any process other than the last does an exit
    (intentional
17  * or unintentional).
18  */
19
20  // Scott's comments: In the original the first sem was the actual
    semaphore.
21  // Now the 3rd though num_sems + 2 are the actual semaphores.
    This should be
22  // transparent to the caller, who should assume the indices of
    the actual sems
23  // run from 0 to n - 1. Much work is also done in the next 20
    lines to allow
24  // the same code to be used in ring 2 or ring 3. Before
    attempting to trace
25  // this code make sure you understand the following defines
    sections. The
26  // sem_open command for ring 2 is only designed to work with the
    TPS. This
27  // is not portable code since the key is parsed to determine the
    (sl,il)
28  // of the semaphore segment according to the following formula:
29  //  key = base + sl * 10 + il; Since we are operating only from
    sl0 - sl2
30  //   and always at il3. The sl is calculated as (key/10)%10, and
    il
31  //   is calculated as key%10. See k_get_sl and k_get_il below.
32  // There is an issue using the SEM_UNDO flag when forking
    children that I
33  // don't understand yet.
34
35 #include "msem.h"
36
37 #ifdef OSS_OPTION  // compiling ring 2 application.
38
```

```
39 #define semop(a,b,c) semaphore_operation((a),(b),(c),APPL_RING)
40 #define semctl(a,b,c,d) semaphore_control((a), (b), (c), (d),
   APPL_RING)
41 void perr_sys( char * mssg, error_code err );
42 #define err_sys(a) perr_sys((a), err)
43 #define ERROR_TEST err != NO_ERROR
44 #define ERROR_CODE error_code
45 static void * U_ZERO;            // ring 2 needs a different
   control_ds type.
46 #else
47
48 #define sem_operation struct sembuf
49 void err_sys( char * mssg );
50 extern int semget( key_t, int, int );
51 extern int semop( int, sem_operation*, int );
52 extern int semctl( int, int, int, union semun arg );
53 #define ERROR_TEST err < 0
54 #define ERROR_CODE int
55 static union semun U_ZERO;
56
57 #endif //OSS_OPTION
58
59
60 #define  BIGCOUNT 10000    /* initial value of process counter */
61 #define   PROC_CNT       1
62 #define   RACE_LK     0
63 #define   SEM_FLG      0    // changed from SEM_UNDO
64 #define err_dump(a) ensure_m(0,(a))
65
66 static int   num_sems;
67 int k_get_sl( key_t key );
68 int k_get_il( key_t key );
69
70
71 /*
72  * Define the semaphore operation arrays for the semop() calls.
73  * These struct provide instructions for the various sems in @
   set.
74  */
75
76 static sem_operation op_lock[2] = {
77    RACE_LK, 0, 0, /* wait for [RACE_LK] (lock) to equal 0 */
78    RACE_LK, 1, SEM_FLG  /* then increment [RACE_LK] to 1 - this
   locks it */
79          /* UNDO to release the lock if processes exits
80             before explicitly unlocking */
81 };
82
83 static sem_operation op_endcreate[2] = {
84    PROC_CNT, -1, SEM_FLG,/* decrement [1] (proc counter) with undo
   on exit */
85          /* UNDO to adjust proc counter if process exits
86             before explicitly calling sem_close() */
87    RACE_LK, -1, SEM_FLG /* then decrement [RACE_LK] (lock) back to
   0 */
88 };
89
```

```
 90 static sem_operation op_open[1] = {
 91    PROC_CNT, -1, SEM_FLG   /* decrement [1] (proc counter) with
    undo on exit */
 92 };
 93
 94 static sem_operation    op_close[3] = {
 95    RACE_LK, 0, 0, /* wait for [2] (lock) to equal 0 */
 96    RACE_LK, 1, SEM_FLG, /* then increment [2] to 1 - this locks it
    */
 97    PROC_CNT, 1, SEM_FLG /* then increment [1] (proc counter) */
 98 };
 99
100 static sem_operation op_unlock[1] = {
101    RACE_LK, -1, SEM_FLG /* decrement [2] (lock) back to 0 */
102 };
103
104 static sem_operation op_op[1] = {
105    0, 99, SEM_FLG /* decrement or increment [0] with undo on exit
    */
106          /* the 99 is set to the actual amount to add
107             or subtract (positive or negative) */
108 };
109
110
       /*************************************************************
    *****************
111  * Create a set of semaphores with a specified initial value.
112  * If the semaphore already exists, we don't initialize it (of
    course).
113  * We return the semaphore ID if all OK, else -1.
114  */
115
116 int
117 sem_create(key, initval, num_sem)
118 key_t key;
119 int   initval; /* used if we create the semaphore */
120 int num_sem;
121 {
122    int debug_on = 0;
123
124    ERROR_CODE err = 0;
125
126    int      id, semval;
127    union semun sem_arg;
128
129    num_sems = num_sem + 2; // for RACE_LK and PROC_CNT.
130
131    ushort *init_array = malloc( sizeof(ushort) * num_sems );
132    ensure(init_array != NULL );
133    for( int idx = 0; idx < num_sems; idx++ )
134    {
135       init_array[idx] = 0;
136  }
137
138    if (key == IPC_PRIVATE)
139       return(-1); /* not intended for private semaphores */
140
```

```
141     else if (key == (key_t) -1)
142        return(-1); /* probably an ftok() error by caller */
143
144 again:
145 #ifdef OSS_OPTION
146  *((ushort *)U_ZERO) = 0;
147     access_ma maccess;
148     access_da daccess;
149
150     maccess.security_level = k_get_sl(key);
151     maccess.integrity_level = k_get_il(key);
152     dbugd(debug_on, "maccess.security_level = ",
    maccess.security_level);
153     dbugd(debug_on, "maccess.integrity_level = ",
    maccess.integrity_level);
154
155     maccess.integrity_categories = 0;
156     maccess.security_categories[0] = 0;
157     maccess.security_categories[1] = 1;
158
159     daccess.owner_perms = READ_MODE | WRITE_MODE;
160     daccess.group_perms = READ_MODE | WRITE_MODE;
161     daccess.other_perms = READ_MODE | WRITE_MODE;
162
163     err = get_semaphore( key, &maccess, APPL_RING, num_sems,
164              0666 | IPC_CREAT, daccess, &id );
165     if(ERROR_TEST)
166     {
167        err_sys("get_semaphore: error ");
168        return(-1);
169     }
170
171 #else // not OSS_OPTION
172     U_ZERO.val = 0;  // initialize U_ZERO for use by all other
    calls.
173
174     if ( (id = semget(key, num_sems, 0666 | IPC_CREAT)) < 0)
175        return(-1); /* permission problem or tables full */
176
177 #endif // OSS_OPTION
178
179     dbugd( debug_on, "seget returned id = ", id );
180     sem_arg.array = init_array;
181 #ifdef OSS_OPTION
182     void * v_ptr = init_array;
183     err = semctl( id, num_sems, SETALL, v_ptr );
184 #else
185     err = semctl( id, num_sems, SETALL, sem_arg );
186 #endif // OSS_OPTION
187
188     if(ERROR_TEST)
189     {
190        err_sys("SETALL failed in sem_create");
191     }
192     dbugd( debug_on, "num_sems = ", num_sems );
193  free(init_array);
194     /*
```

```
195      * When the semaphore is created, we know that the value of all
196      * SEM_SET_SIZE members is 0.
197      * Get a lock on the semaphore by waiting for [2] to equal 0,
198      * then increment it.
199      *
200      * There is a race condition here.  There is a possibility that
201      * between the semget() above and the semop() below, another
202      * process can call our sem_close() function which can remove
203      * the semaphore if that process is the last one using it.
204      * Therefore, we handle the error condition of an invalid
205      * semaphore ID specially below, and if it does happen, we just
206      * go back and create it again.
207      */
208
209     err = semop(id,  &op_lock[0], 2);
210 #ifdef OSS_OPTION
211     if( err == MESSAGE_RECEIVED ) goto again;
212     else if(ERROR_TEST) err_sys("can't lock");
213
214 #else // NOT OSS_OPTION
215     if (ERROR_TEST) {
216        if (errno == EINVAL)
217           goto again;
218        err_sys("can't lock");
219     }
220
221 #endif //OSS_OPTION
222
223     /*
224      * Get the value of the process counter.  If it equals 0,
225      * then no one has initialized the semaphore yet.
226      */
227     err = semctl(id, PROC_CNT, GETVAL, U_ZERO);
228  semval = err;
229     if (ERROR_TEST)
230        err_sys("can't GETVAL");
231     dbugd( debug_on, "creat_sem: PROC_CNT value is ", semval );
232
233     if (semval == 0) {
234        /*
235         * We could initialize by doing a SETALL, but that
236         * would clear the adjust value that we set when we
237         * locked the semaphore above.  Instead, we'll do 2
238         * system calls to initialize [0] and [1].
239         */
240
241        for(int idx = 2; idx < num_sems; idx++)
242        {
243           err = semctl(id, idx, SETVAL, U_ZERO);
244           if (ERROR_TEST)
245              err_sys("can't SETVAL");
246        }
247
248        sem_arg.val = BIGCOUNT;
249 #ifdef OSS_OPTION
250        void * v_ptr = &(sem_arg.val);
251        err = semctl(id, PROC_CNT, SETVAL, v_ptr);
```

```
252 #else // not OSS_OPTION
253       err = semctl(id, PROC_CNT, SETVAL, sem_arg);
254 #endif //OSS_OPTION
255       if (ERROR_TEST)
256          err_sys("can SETVAL[PROC_CNT]");
257    }
258
259    /*
260     * Decrement the process counter and then release the lock.
261     */
262    err = semop(id, &op_endcreate[0], 2);
263    if (ERROR_TEST)
264       err_sys("can't end create");
265
266    return(id);
267 }
268
269
       /************************************************************
    *****************
270  * Open a semaphore that must already exist.
271  * This function should be used, instead of sem_create(), if the
    caller
272  * knows that the semaphore must already exist.  For example a
    client
273  * from a client-server pair would use this, if its the server's
274  * responsibility to create the semaphore.
275  * We return the semaphore ID if all OK, else -1.
276  */
277
278 int
279 sem_open(key)
280 key_t key;
281 {
282    int debug_on = 0;
283    ERROR_CODE err = 0;
284
285    dbugd( debug_on, "sem_open: key = ", key );
286
287    int   id;
288
289    if (key == IPC_PRIVATE)
290       return(-1); /* not intended for private semaphores */
291
292    else if (key == (key_t) -1)
293       return(-1); /* probably an ftok() error by caller */
294
295 #ifdef OSS_OPTION
296  access_ma maccess;
297  access_da daccess;
298
299  maccess.security_level = k_get_sl(key);
300  maccess.integrity_level = k_get_il(key);
301  dbugd(debug_on, "k_get_sl(key) = ", k_get_sl(key));
302  dbugd(debug_on, "k_get_il(key) = ", k_get_il(key));
303  maccess.integrity_categories = 0;
304  maccess.security_categories[0] = 0;
```

```
305  maccess.security_categories[1] = 0;
306
307  daccess.owner_perms = READ_MODE | WRITE_MODE;
308  daccess.group_perms = READ_MODE | WRITE_MODE;
309  daccess.other_perms = READ_MODE | WRITE_MODE;
310
311  err = get_semaphore( key, &maccess, APPL_RING, num_sems,
312           0, daccess, &id );
313    if(err != NO_ERROR )
314    {
315       print_error("get_semaphore: error ", err );
316       return(-1);
317    }
318 #else // not OSS_OPTION
319
320    if ( (id = semget(key, num_sems, 0)) < 0)
321       return(-1); /* doesn't exist, or tables full */
322 #endif // OSS_OPTION
323
324    /*
325     * Decrement the process counter.  We don't need a lock
326     * to do this.
327     */
328    err = semop(id, &op_open[0], 1);
329    if (ERROR_TEST)
330       err_sys("can't open");
331
332    dbugd( debug_on, "sem_open: returning id = ", id );
333
334    return(id);
335 }
336
337
        /*************************************************************
    *****************
338  * Remove a semaphore.
339  * This call is intended to be called by a server, for example,
340  * when it is being shut down, as we do an IPC_RMID on the
    semaphore,
341  * regardless whether other processes may be using it or not.
342  * Most other processes should use sem_close() below.
343  */
344
345 void sem_rm(id)
346 int   id;
347 {
348    ERROR_CODE err = 0;
349    err = semctl(id, 0, IPC_RMID, U_ZERO);
350    if (ERROR_TEST)
351       err_sys("can't IPC_RMID");
352 }
353
354
        /*************************************************************
    *****************
355  * Close a semaphore.
```

```
356  * Unlike the remove function above, this function is for a
    process
357  * to call before it exits, when it is done with the semaphore.
358  * We "decrement" the counter of processes using the semaphore,
    and
359  * if this was the last one, we can remove the semaphore.
360  */
361
362 void sem_close(id)
363 int   id;
364 {
365    int debug_on = 0;
366    ERROR_CODE err = 0;
367    dbugd( debug_on, "sem_close: id = ", id );
368    ensure( id >= 0 );
369    register int   semval;
370
371    /*
372     * The following semop() first gets a lock on the semaphore,
373     * then increments [1] - the process counter.
374     */
375    err = semop(id, &op_close[0], 3);
376    if(ERROR_TEST)
377       err_sys("can't semop");
378
379    /*
380     * Now that we have a lock, read the value of the process
381     * counter to see if this is the last reference to the
382     * semaphore.
383     * There is a race condition here - see the comments in
384     * sem_create().
385     */
386    err = semctl(id, PROC_CNT, GETVAL, U_ZERO);
387    if (ERROR_TEST)
388       err_sys("can't GETVAL");
389    semval = err;
390
391    if (semval > BIGCOUNT)
392       err_dump("sem[1] > BIGCOUNT");
393    else if (semval == BIGCOUNT)
394       sem_rm(id);
395    else
396    {
397       err = semop(id, &op_unlock[0], 1);
398       if (ERROR_TEST)
399          err_sys("can't unlock");    /* unlock */
400    }
401 }
402
403
       /************************************************************
    ****************
404  * Wait until a semaphore's value is greater than 0, then
    decrement
405  * it by 1 and return.
406  * Dijkstra's P operation.  Tanenbaum's DOWN operation.
407  */
```

```
408
409 void sem_wait(id, idx)
410 int   id;
411 int idx;
412 {
413    sem_op(id, idx, -1);
414 }
415
416
       /*************************************************************
   *****************
417  * Increment a semaphore by 1.
418  * Dijkstra's V operation.  Tanenbaum's UP operation.
419  */
420
421 void sem_signal(id, idx)
422 int   id;
423 int idx;
424 {
425    sem_op(id, idx, 1);
426 }
427
428
       /*************************************************************
   *****************
429  * General semaphore operation.  Increment or decrement by a user-
   specified
430  * amount (positive or negative; amount can't be zero).
431  */
432
433 void sem_op(id, idx, value)
434 int   id;
435 int idx;
436 int   value;
437 {
438    int debug_on = 0;
439
440    ERROR_CODE err = 0;
441    dbugd( debug_on, "entered sem_op id = ", id );
442    dbugd( debug_on, "entered sem_op idx = ", idx );
443    dbugd( debug_on, "entered sem_op value = ", value );
444  dbugd( debug_on, "entered sem_op num_sems = ", num_sems );
445
446    if ( (op_op[0].sem_op = value) == 0)
447       err_sys("can't have value == 0");
448
449    op_op[0].sem_num = idx + 2; // to compenstate for having two
   extra sems in 0 and 1
450    err = semop(id, &op_op[0], 1);
451    if (ERROR_TEST)
452       err_sys("sem_op error");
453 }
454
455
456 // Private utility functions.
457
458 #ifdef OSS_OPTION
```

```
459
460 void perr_sys( char * mssg, ERROR_CODE err )
461 {
462    print_error(mssg, err);
463    exit(-1);
464 }
465
466 #else not OSS_OPTION
467
468 void err_sys( char * mssg )
469 {
470       perror(mssg);
471       exit(-1);
472 }// end err_sys
473
474 #endif // OSS_OPTION
475
476 int k_get_sl( key_t key )
477 {
478       return( (key/10) % 10 );
479 }
480
481 int k_get_il( key_t key )
482 {
483       return( key%10 );
484 }
485
486
```

```
 1 // File: priv_util.c
 2 // Author: Scott Heller
 3 // Date:   27 Feb 99
 4 // Purpose: One stop shopping for gaining the privileges needed to
 5 //  execute as a Secure Session Server. App needs to first be
   installed
 6 //  with privileges using "tp_edit" with administrator access.
 7
 8 #include "priv_util.h"
 9
10 void set_priv( ushort priv )
11 {
12       int debug_on = 0;
13
14 #ifdef OSS_OPTION
15       dbugd( debug_on, "set_priv: setting privileges = ", priv );
16       (void)set_privilege( priv );
17       dbug( debug_on, "set_priv: set privileges = ");
18
19 #else
20       dbug( debug_on, "set_priv: ring 3 - doing nothing");
21
22 #endif // OSS_OPTION
23
24 }// end set_priv()
25
26 ushort  enable_priv(void)
27 {
28       int debug_on = 0;
29       ushort         old_priv = 0, new_priv = 0;
30
31       dbug( debug_on, "enable_priv: entered");
32 #ifdef OSS_OPTION
33
34       process_status proc_stat;
35
36       if( get_process_status(0, &proc_stat) == NO_ERROR)
37       {
38
   if(proc_stat.max_privilege.privilege.simple_security_exempt &&
39
   proc_stat.max_privilege.privilege.simple_integrity_exempt &&
40
   proc_stat.max_privilege.privilege.security_star_property_exempt &&
41
   proc_stat.max_privilege.privilege.integrity_star_property_exempt )
42          {
43             new_priv |= SIMPLE_SECURITY_EXEMPT;
44             new_priv |= SIMPLE_INTEGRITY_EXEMPT;
45             new_priv |= SECURITY_STAR_PROPERTY_EXEMPT;
46             new_priv |= INTEGRITY_STAR_PROPERTY_EXEMPT;
47          } else {
48             dbug( debug_on, "enable_priv: SET PROPER PRIVS USING
   TP_EDIT");
49          }// end if
50
51       }// end if
```

```
52
53          old_priv = add_privilege( new_priv );
54  #else // not a ring 2 application
55          dbug( debug_on, "enable_priv: not in ring 2: doing
    nothing");
56
57  #endif // OSS_OPTION
58
59          return old_priv;
60  }// end enable_priv()
61
62  // return -1 on error, 0 on success.
63  int get_current_level( struct level_struct * result )
64  {
65          int debug_on = 0;
66          int error = false, status = 0;
67
68  #ifdef OSS_OPTION
69          access curr_a;
70
71          dbug( debug_on, "enable_priv: not in ring 2: doing
    nothing");
72          // need <access.h>, and <tcb_gates.h>
73          if( get_process_access( 0, &curr_a ) == NO_ERROR )
74          {
75             result->sl = curr_a.ma.security_level;
76             dbugd( debug_on, "get_current_level: sl = ", result->sl
    );
77             result->il = curr_a.ma.integrity_level;
78             dbugd( debug_on, "get_current_level: il = ", result->il
    );
79          } else {
80             error = true;
81          }// end if
82
83  #else // This is a ring 3 application.
84          // need <level.h> and -lcass
85          access_ma curr_ma;
86
87          if( getlevel( NULL, &curr_ma ) )
88          {
89             error = true;
90          } else {
91             // getlevel succeeded.
92             result->sl = curr_ma.security_level;
93             result->il = curr_ma.integrity_level;
94             dbugd( debug_on, "get_current_level: sl = ", result->sl
    );
95             dbugd( debug_on, "get_current_level: il = ", result->il
    );
96
97          }// end if
98
99  #endif // OSS_OPTION
100         if(error) {
101            result->sl = -1;
102            result->il = -1;
```

```
103             status = -1;
104         }
105
106         return status;
107
108 }// end get_current_level()
109
110
111 key_t calc_key( int base )
112 {
113         int debug_on = 0;
114         key_t key = -1;
115
116         struct level_struct lvl;
117
118         // get the current level for the key calculation
119         if( get_current_level( &lvl ) )
120         {
121             dbug(debug_on, "calc_key: error occured");
122         } else {
123             // calculate the key.
124             key = lvl.sl * 10 + lvl.il + base;
125
126         }// end if
127
128         dbugd( debug_on, "calc_key: key = ", key );
129
130         return key;
131 }// end calc_key()
132
```

```
 1 // File: shm.c
 2 // Author: Scott Heller
 3 // Date:   17 Feb 99
 4 // Purpose: Create and manipulate a shared memory segment
 5 // between ring 2 and 3.
 6
 7
 8 #include "shm.h"
 9
10
11 // create and attach to a shared memory segment
12 // make this segment availible to the protocol server
13 // PRE: addr == NULL;
14 // Return: shared memory ID. addr = <first address of shared mem>
15 // size of the shm seg is sizeof(struct in_buff_struct);
16 // NOTE: Do not loose the addr. Needed for removal of shm segment.
17 // Todo: May have to set the shm_perm.mode for ug+rw
18 int get_shm(key_t key, void ** addr,  size_t size )
19 {
20       int debug_on = 0;
21
22       int result = 0;
23
24       //ensure(addr == NULL );
25
26       dbug(debug_on, "shm:get_shm: entered" );
27       dbugd( debug_on, "shm:get_shm: with key = ", key );
28
29       result = shmget( key, size, SHM_PERM | IPC_CREAT );
30
31       dbugd(debug_on, "shmget called shmid = ", result );
32       if( result != -1 && debug_on )
33       {
34          struct shmid_ds shm_ds;
35          if( !shmctl( result, IPC_STAT, &shm_ds ))
36          {
37             printf("shm_ds: mode %d, size %d, creator %d\n",
38                   shm_ds.shm_perm.mode,   shm_ds.shm_segsz,
39                shm_ds.shm_cpid );
40          } else {
41             perror("shmctl failed to get IPC_STAT");
42          } //end if
43       }// end if
44
45
46       if(result != -1 )
47       {
48          // shmget successful now attach the shm segment.
49          *addr = shmat(result, (void *)0, 0 );
50          dbugd( debug_on, "shmat:addr = ", (int)addr);
51          if(addr == (void *) -1 )
52          {
53             // shmat failed
54             perror("shm: shmat failed");
55             result = -1;
56
57             // shmctl rtns 0 if successful -1 on error
```

```
58              if(shmctl(result, IPC_RMID, (struct shmid_ds *)0 ) )
59              {
60                 perror("shm:shmctl IPC_RMID failed");
61              }
62           }// end if
63
64        }// end if
65
66        dbugd( debug_on, "get_shm exiting shmid = ", result );
67        return result;
68
69 }// end get_shm
70
71
72 // return a pointer to a shm segment from a shmid.
73 void *attach_shm( int shmid )
74 {
75        int debug_on = 0;
76
77        dbugd( debug_on, "attach_shm: entered with shmid = ", shmid
   );
78        void * addr = shmat( shmid, (char *)0, 0 );
79
80        return addr;
81 }// end attach_shm
82
83
84
85 // detach and remove shared memory segment from the system
86 // this must be done every time prior to exit being called
87 void remove_shm(int shm_id, void * addr )
88 {
89        int debug_on = 0;
90        dbug(debug_on, "shm:remove_shm:entered" );
91
92        // detach shared mem segment
93        if(shmdt(addr) )
94        {
95           perror("shm:remove_shm:shmdt failed");
96        }
97
98        // dispose of shared mem segment
99        if(shmctl(shm_id, IPC_RMID, (struct shmid_ds *)0 ) )
100       {
101          perror("shm:shmctl IPC_RMID failed");
102       }
103
104 }// end remove_shm
105
106
107
```

```
 1 // File: shm_struct.c
 2 // Author: Scott Heller
 3 // Date:  22 Feb 1999
 4 // Purpose: Provide interface for connection shm. One shm segment
 5 //    per level.
 6
 7 #include "shm_struct.h"
 8
 9 // This is only called from ring 3 during protocol server's
10 // call to socket(). If used by ring 2 app will cause undesired
   side affects.
11 int init_shm_hdr( struct shm_hdr ** shm_ptr )
12 {
13       int debug_on = 0;
14
15       key_t level_key;
16       key_t level_sem_key;
17       int result = 1, shm_id = 0;
18
19       dbug( debug_on, "Entered init_shm_hdr" );
20
21       // get_sl * 10 + get_il + SHM_STRUCT_BASE_KEY
22       level_key = calc_key( SHM_STRUCT_BASE_KEY );
23
24       shm_id = get_shm(level_key,(void *)shm_ptr,sizeof(struct
   shm_hdr));
25
26       if(shm_id == -1)
27       {
28          perror("init_shm_hdr:get_shm returned error");
29          result = -1;
30       } else {
31          dbugd( debug_on, "About to assign shm_ptr->shm_hdr_shmid
   = ", shm_id );
32          // shm_id is valid.
33          (*shm_ptr)->shm_hdr_shmid = shm_id;
34          level_sem_key = LEVEL_SEM_KEY_BASE;
35          level_sem_key = calc_key( level_sem_key );
36
37          // create semaphores for connections
38          (*shm_ptr)->conn_semid =
   sem_create(level_sem_key,0,MAX_OPEN_CONN );
39
40          if( (*shm_ptr)->conn_semid == -1)
41          {
42             perror("init_shm_hdr:failed to create conn
   semaphores");
43             ss_cleanup(*shm_ptr);
44             result = -1;
45          }
46       }
47
48       // initialize the listen queue
49       lq_init( &((*shm_ptr)->lq) );
50
51       // set all connections available and initialize buffers.
52       for(int idx = 0; idx < MAX_OPEN_CONN ; idx++ )
```

```
53        {
54           (*shm_ptr)->conn[idx].in_use = 0;
55           init_buffer( &( (*shm_ptr)->conn[idx].to_svr_buff ) );
56           init_buffer( &( (*shm_ptr)->conn[idx].to_cli_buff ) );
57        }
58
59        if(debug_on) printf("init_shm_hdr:shm_ptr address is: %x\n",
   (int)shm_ptr );
60        if(debug_on) printf("init_shm_hdr:shm_ptr points to addr:
   %x\n",
61           (int)*shm_ptr );
62        dbugd( debug_on, "init_shm_hdr: exiting result = ", result
   );
63
64        return result;
65
66 }// end init_shm_hdr()
67
68 // currently never called. Not a big issue since, for demo
   purposes
69 // only up to 3 shm segments are ever created and the same ones
70 // are always reused.
71 void ss_cleanup( struct shm_hdr * shm_ptr )
72 {
73        int debug_on = 1;
74        dbug(debug_on, "EXECUTING ss_cleanup");
75
76        remove_shm( shm_ptr->shm_hdr_shmid, (void *)shm_ptr );
77        sem_rm( shm_ptr->conn_semid );
78
79 }// end ss_cleanup()
80
81
82 int ss_read(int fd, struct shm_hdr * shmhdr, char *buff, int
   nbytes )
83 {
84        int debug_on = 0;
85
86        int n = 0;
87
88        dbugd( debug_on, "ss_read: fd = ", fd );
89        if(debug_on) print_buff_queue( &(shmhdr-
   >conn[fd].to_svr_buff));
90
91        // if in_use read, other wise return error.
92        if( shmhdr->conn[fd].in_use )
93           n = buff_io_read(&(shmhdr->conn[fd].to_svr_buff), buff,
   nbytes);
94        else
95           n = -1;
96
97        return n;
98 }// end ss_read
99
100 int ss_write(int fd, struct shm_hdr *shmhdr, const char *data, int
   nbytes )
101 {
```

```
102       int n = 0;
103       if( shmhdr->conn[fd].in_use )
104          n = add_data_part( data, &(shmhdr->conn[fd].to_cli_buff),
    nbytes);
105       else
106          n = -1;
107
108       return n;
109
110 }// end ss_write()
111
112 int ss_read_fm_svr(int fd, struct shm_hdr *shmhdr, char *buff, int
    nbytes)
113 {
114       return(buff_io_read(&(shmhdr->conn[fd].to_cli_buff), buff,
    nbytes));
115 }// end ss_read_fm_svr
116
117
118 int ss_write_to_svr(int fd, struct shm_hdr *shmhdr, const char
    *data, int nbytes)
119 {
120       int n = 0;
121       n = add_data_part( data, &(shmhdr->conn[fd].to_svr_buff),
    nbytes);
122       return n;
123 }// end ss_write_to_svr()
124
125 void ss_close( int fd, struct shm_hdr *shmhdr )
126 {
127       int debug_on = 0;
128 #ifdef DEMO
129       debug_on = 1;
130 #endif //DEMO
131
132       dbugd( debug_on, "ss_close: closing idx = ", fd );
133
134       shmhdr->conn[fd].in_use = 0;
135
136 }// end ss_close()
137
138 int ss_data_avail( int idx, struct shm_hdr *shmhdr )
139 {
140       int debug_on = 0;
141       int n = 0;
142
143       dbugd( debug_on, "ss_data_avail: entered idx = ", idx );
144       ensure( idx >= 0 && idx < MAX_OPEN_CONN );
145
146       if(debug_on ) print_buff_queue( &(shmhdr-
    >conn[idx].to_svr_buff ));
147
148       n = num_char( &(shmhdr->conn[idx].to_svr_buff ));
149       dbugd( debug_on, "ss_data_avail: I think there are n bytes
    => ", n );
150
151       return ( n );
```

```
152
153 }// end ss_data_avail()
154
155
156 int ss_space_avail( int idx, struct shm_hdr *shmhdr )
157 {
158       int debug_on = 0;
159       ensure( idx >= 0 && idx < MAX_OPEN_CONN );
160
161       return( bytes_free( &(shmhdr->conn[idx].to_cli_buff) ));
162
163 }// end ss_space_avail()
164
165 int ss_socket_error( int idx, struct shm_hdr *shmhdr )
166 {
167       int debug_on = 0;
168       ensure( idx >= 0 && idx < MAX_OPEN_CONN );
169       dbugd( debug_on, "ss_socket_error: entered for connection ",
    idx );
170       int result = shmhdr->conn[idx].in_use;
171       result = (result == 1 )? 0 : 1;
172       dbugd( debug_on, "ss_socket_error: returning ", result );
173
174       return( result );
175
176 }// end ss_socket_error()
177
178
179 int ss_block_on_lq( struct shm_hdr *shmhdr )
180 {
181       int debug_on = 0;
182       dbug( debug_on, "ss_block_on_lq: entered" );
183       int new_socket = lq_remove_item( &(shmhdr->lq) );
184
185       return new_socket;
186 }// end ss_block_on_lq
187
188 void ss_copy_cli_buff( struct shm_hdr *shmhdr, int idx,
189                           struct in_buff_struct *from )
190 {
191
192     (void) memcpy( (void *)&(shmhdr->conn[idx].to_cli_buff) ,
193                    (void *)from, sizeof(struct in_buff_struct) );
194
195 }// end ss_copy_cli_buff
196
197
198 int ss_request_connection( struct shm_hdr *shmhdr )
199 {
200       int debug_on = 0;
201       dbug(debug_on, "ss_request_connection entered");
202
203       int idx = 0, new_conn = -1;
204
205       while( idx < MAX_OPEN_CONN )
206       {
207          dbug( debug_on, "checking shmhdr->conn[idx].in_use" );
```

```
208          if(shmhdr->conn[idx].in_use)
209          {
210             dbugd( debug_on, "connection busy: ", idx );
211             idx++;
212          } else
213             break;
214       }// end while
215
216       dbugd( debug_on, "ss_request_conn: while loop finishted idx
    = ", idx );
217       if( idx < MAX_OPEN_CONN )
218       {
219          new_conn = idx;
220          init_buffer( &(shmhdr->conn[new_conn].to_svr_buff) );
221          init_buffer( &(shmhdr->conn[new_conn].to_cli_buff) );
222          if( ! lq_add_item( new_conn, &(shmhdr->lq) ) ) new_conn =
    -1;
223       } else {
224          new_conn = -1;
225          dbug( debug_on, "No connections avail, try again later");
226       }
227
228       dbugd( debug_on, "ss_request_connection returning => ",
    new_conn );
229       shmhdr->conn[new_conn].in_use = 1;
230       return new_conn;
231
232 }// end ss_request_connection
233
234
235 int ss_xfer_skt_buff( struct shm_hdr *shmhdr, int pskfd, int
    sockfd )
236 {
237       int debug_on = 0;
238       int n = 0;         // number of bytes transfered.
239
240       dbugd( debug_on, "ss_xfer_skt_buff: to conn => ", pskfd);
241       // enter cs
242       n = get_data( sockfd, &(shmhdr->conn[pskfd].to_svr_buff ));
243       // exit cs
244       dbugd( debug_on, "ss_xfer_skt_buff: get_data reports nbytes
    added",
245          n );
246
247       return n;
248 }// end ss_xfer_skt_buff
249
250 int ss_xfer_buff_skt( struct shm_hdr *shmhdr, int pskfd, int
    sockfd )
251 {
252       int debug_on = 0;
253
254       int n = 0;         // number of bytes transfered
255       char t_buff[INBUFFSIZE];
256
257       //enter cs
258       n = ss_read_fm_svr(pskfd, shmhdr, t_buff , INBUFFSIZE - 1 );
```

```
259         dbugd( debug_on, "ss_xfer_buff_skt: ss_read_fm_svr read
    nbytes = ",
260            n );
261         dbug( debug_on, t_buff );
262
263         if( n > 0 )
264            n = Writen( sockfd, t_buff, n );
265         else
266            dbug( debug_on, "ss_read_fm_svr returned error in
    ss_xfer_buff_skt");
267
268         return n;
269
270 }// end ss_xfer_buff_skt()
271
272 int ss_get_hdr( struct shm_hdr ** addrOf_shm_hdr,
273                    struct user_ia_struct *ia_data )
274 {
275         int debug_on = 0;
276
277         int        shmid = -1;
278         key_t key;
279
280         if( ia_data == NULL )
281         {
282            dbug( debug_on, "ss_get_hdr: ia_data == NULL" );
283            key = calc_key( SHM_STRUCT_BASE_KEY );
284            shmid = get_shm(key,(void **)addrOf_shm_hdr,sizeof(struct
    shm_hdr));
285         } else {
286            dbugd( debug_on, "ss_get_hdr: ia_data->sl = ", ia_data-
    >sl );
287            dbugd( debug_on, "ss_get_hdr: ia_data->il = ", ia_data-
    >il );
288 #ifdef OSS_OPTION
289            access_ma maccess;
290            access_da daccess;
291            error_code err = NO_ERROR;
292            _near void * far_addr;
293
294            dbug( debug_on, "*ss_get_hdr: far_addr declared" );
295
296
297            maccess.security_level = (utiny)ia_data->sl;
298            maccess.integrity_level = (utiny)ia_data->il;
299            maccess.integrity_categories = 0;
300            maccess.security_categories[0] = 0;
301            maccess.security_categories[1] = 0;
302
303            daccess.owner_perms = READ_MODE | WRITE_MODE; // rw
304            daccess.group_perms = READ_MODE | WRITE_MODE; // rw
305            daccess.other_perms = READ_MODE | WRITE_MODE; // rw
306            //daccess.acl_id[0] = 0;   // see TFPM access(5)
307            //daccess.acl_perms[0] = 0;
308            key = ia_data->sl * 10 + ia_data->il +
    SHM_STRUCT_BASE_KEY;
309
```

```
310
311          err = get_shared_memory( key, &maccess, APPL_RING,
    APPL_RING,
312             sizeof( struct shm_hdr ), 0, daccess, &shmid );
313          if( err != NO_ERROR )
314          {
315             dbugd( debug_on, "get_shared_memory shmid = ", shmid
    );
316             print_error( "ss_get_hdr:get_shared_memory error", err
    );
317             shmid = -1;
318             perror("ss_get_hdr:get_shared_memory error");
319          }// end if
320
321          dbugd(debug_on, "ss_get_hdr: get_shared_memory returned
    shmid =",
322             shmid );
323
324          err = attach_shared_memory( shmid, 0, NULL, false,
    &(far_addr) );
325          if( err != NO_ERROR )
326          {
327             dbugd( debug_on, "attach_shared_memory shmid = ",
    shmid );
328             print_error( "ss_get_hdr:attach_shared_memory error",
    err );
329             shmid = -1;
330             perror("ss_get_hdr:get_shared_memory error");
331             exit(-1);
332          }// end if
333          if(debug_on)
334          {
335             printf("far_addr = %8.8x \n",
336                far_addr );
337          }
338          if( far_addr == 0 ) exit (-1);
339          *addrOf_shm_hdr = (struct shm_hdr *)( far_addr);
340
341 #else // NOT USING OSS_OPTION
342          key = ia_data->sl * 10 + ia_data->il +
    SHM_STRUCT_BASE_KEY;
343          shmid = get_shm(key,(void **)addrOf_shm_hdr,sizeof(struct
    shm_hdr));
344 #endif //OSS_OPTION
345       }// end if
346
347       dbugd( debug_on, "ss_get_hdr: using key = ", key );
348       dbugd( debug_on, "ss_get_hdr: returning shmid  = ", shmid );
349
350       return(shmid);
351
352 }// end ss_get_hdr
353
354 void ss_detach_hdr( struct shm_hdr * shm_hdr )
355 {
356       if( shmdt( (void *)shm_hdr ) )
357       {
```

```
358             perror("ss_detach_hdr: shmdt() failed");
359         }
360 }// end ss_detach_hdr
361
362
363
```

```
 1 // File:    tps.c
 2 // Author:  Scott D. Heller & Susan Bryer-Joyner
 3 // Date:    28 January 1999
 4 // Purpose: Main() for the Trusted Path Server
 5
 6
 7 #include    <errno.h>
 8 #include    <stdio.h>
 9 #include    <types.h>
10 #include    <sys/socket.h>
11 #include    <netinet/in.h>
12 #include    <string.h>
13 #include    <unistd.h>
14 #include    <stdlib.h>
15 #include    <sys/byteorder.h>    // for htonl and htons
16
17 #ifdef   OSS_OPTION              // must set -DOSS_OPTION when
   compiling TPS
18                                  //   for ring 2.
19 #include    <stop/privileges.h>  // for granting privileges
20                                  // must also use tp_edit
21 #include    <stop/tcb_gates.h>   // used for fork_process
22 #include    <error_code.h>       // for suspend event
23 #include    <message.h>          // for suspend event
24 #endif //OSS_OPTION
25
26 #include    <fcntl.h>
27
28 #include    "util.h"
29 #include    "tps_util.h"
30 #include   "user_ia.h"
31 #include    "cdb.h"
32 #include   "buff_io.h"
33
34 #ifndef OSS_OPTION  // if NOT OSS_OPTION set
35 #define     fork_process() fork()
36 extern      int fork();
37 #else   // OSS_OPTION is set.
38 #define     sleep(a)
   suspend_event(NO_EVENT,(a)*ONE_SECOND,0,NULL,NULL,NULL)
39 #endif //OSS_OPTION
40
41
42 extern int fcntl( int, int, int ); // to eliminate warning.
43
44 int main ( int argc, char **argv )
45 {
46
47    int   debug_on = 1;
48
49    int   listenfd    = 0,
50          connfd      = 0,
51          clilen      = 0,
52          testBind    = 0,
53          cap         = 0,  // Controlling Active Process
54          sak_attempts = 0, // note signal handler needs to reset
   this.
```

```
55          hw_id       = -1,
56          flag        = 0;
57
58    struct in_buff_struct *buffer;
59    struct user_ia_struct ia_data;
60
61    int cdb_size = 0;
62    cdb_size = init_cdb( );
63
64    dbugd(debug_on, "Start execution.TPS pid = ", getpid() );
65
66    struct   sockaddr_in cliaddr, servaddr;
67    memset( &servaddr, 0, sizeof(servaddr) );
68    servaddr.sin_family      = AF_INET;
69    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
70    servaddr.sin_port        = htons(SERV_PORT);
71
72    listenfd = socket(AF_INET, SOCK_STREAM, 0);
73    ensure( listenfd > -1 );
74
75    testBind = bind(listenfd,(struct
   sockaddr*)&servaddr,sizeof(servaddr));
76    ensure( testBind > -1 );
77
78    dbugd(debug_on, "Listening to port:", SERV_PORT);
79    listen( listenfd, 5 );
80
81    if(debug_on) print_cdb();
82
83    for ( ; ; )
84    {
85       clilen = sizeof( cliaddr );
86
87       // block until connection then accept
88       connfd = accept( listenfd, (struct sockaddr *) &cliaddr,
   &clilen );
89       ensure (connfd > -1);
90
91       // set O_NDELAY for connfd.
92       flag = fcntl( connfd, F_GETFL, 0 );
93       flag |= O_NDELAY;
94       if( fcntl( connfd, F_SETFL, flag ) == -1 )
95       {
96          perror("fcntl set flag failed");
97          exit(-1);
98       }
99
100      // create the child to handle the new connection
101      if( ( fork_process() )  == 0 )
102      {
103         // child process
104         dbugd( debug_on, "TPS Child. pid = ", getpid() );
105
106         // only the parent should use listenfd
107         close(listenfd);
108
```

```
109          // allocate and initialize buffer for storing connfd
    data.
110          dbug( debug_on, "TPS Child: calling malloc struct
    in_buff_struct");
111          buffer = malloc( sizeof( struct in_buff_struct ) );
112          dbug( debug_on, "TPS Child: calling
    init_buffer(buffer)");
113          init_buffer( buffer );
114
115          do { // loop until we have a valid SAK mssg.
116             sak_attempts++;
117             dbug( debug_on, "TPS Child: calling select sleep");
118
119             sleep( 1 ); // 1 second
120
121             dbug(debug_on, "tps.c: Checking SAK Message");
122             if ( (cap = check_SAK(connfd, &hw_id, buffer)) > 0 )
123             {
124                dbugd( debug_on, "TPS rcvd SAS when SSS
    should've.", cap );
125                exit(0);
126
127             } else if( cap == 0 ) {
128                // cap is TPS
129                // make this child the CAP
130                dbugd( debug_on, "child: cap = ", cap );
131                if( update_CDB( hw_id, getpid() ) == -1)
132                {
133                   dbug(debug_on, "hw_id not valid. Exiting");
134                   exit(-1);
135                }
136                if(debug_on) print_cdb( );
137
138                // perform user_IA returns true if valid.
139                ia_data =  user_IA(connfd, buffer);
140                if( ia_data.valid )
141                {
142                   dbug(debug_on, "tps:calling socket_relay");
143                   // finally do all the work.
144                   socket_relay( connfd, buffer, &ia_data );
145                }
146             }
147          } while( cap == -1 && sak_attempts < MAX_SAK_ATTEMPTS );
148
149          dbugd(debug_on, "tps.c:hwid = ", hw_id );
150          // if the current process is the CAPID update CDB
151          // make TPS the CAPID.
152          if( ( cap = get_CAPID( hw_id )) == getpid() )
153          {
154             if( !update_CDB( hw_id, 0 ) )
155                perror("Error updating CDB during cleanup");
156          }
157
158          // allocated above via call to malloc()
159          free( buffer );
160
161          dbugd(debug_on, "Exiting!! :", getpid() );
```

```
162           exit(0);
163       }//end if
164
165       close(connfd); // parent then when finished child
166                      // closes connected socket
167
168    }// end for loop
169
170     return 0;
171
172 }// end main
173
174
```

```
1  // File: tps_util.c
2  // Author: Scott D. Heller & Susan Bryer-Joyner
3  // Date:   28 January 1999
4  // Purpose: Functions used by the Trusted Path Server (tps.c)
5
6  #include "tps_util.h"
7
8
9  // Return:  -1 if not valid SAK mssg else return CAPID
10 // Param:   sockfd: valid socket descriptor
11 //          hw_id : pointer used to return the hardware id of the
   TCBE
12 //          queue : buffer associated with the current connection.
13 // Purpose: Verify the SAS is legitiment. Eventually public-key
   verification
14 //    should occur here.
15 // Note:    A valid SAK mssg, for now, is one the starts:
16 //  "send brk" and is followed by a 1-3 digit hardware ID.
17 int check_SAK(int sockfd, int * hw_id, struct in_buff_struct
   *queue )
18 {
19    int debug_on = 0;
20    dbug(debug_on, "tps_util:check_SAK: beginning");
21
22    char *hwid_buff_ptr;
23
24    int result = -1;
25    int num_read = 0;
26
27    // get hw id from SAK mssg.
28    // get_token allocates memory for and returns char *,
29    // if there was an error (ie. no data avail) NULL returned.
30    hwid_buff_ptr = get_token( sockfd, queue,'\n',  MAXHWID );
31    if( hwid_buff_ptr == NULL ) return (-1);
32
33    // if the message starts out like a SAK mssg.
34    if(   strlen( hwid_buff_ptr ) >= MIN_SAK_LEN &&
35          *(hwid_buff_ptr)     == TELNET_SEND  &&
36          *(hwid_buff_ptr + 1) == TELNET_BRK    )
37    {
38
39       if( sscanf( hwid_buff_ptr + 2, "%d", hw_id ) <= 0 )
40       {
41          printf("check_SAK: failed to convert hwid_buff to int");
42          exit(-1);
43       }//end if
44
45       dbugd(debug_on, "check_SAK:after sscanf hw_id = ", *hw_id );
46
47       // look up hwid in CDB
48       // get CAPID from CDB
49       result = get_CAPID( *hw_id );
50    }
51
52    // memory allocated above by get_token.
53    free( hwid_buff_ptr );
54
```

```
55    // return CAPID
56    return( result );
57
58 }// end check_SAK
59
60
61 int socket_relay( int cli_fd,
62                   struct in_buff_struct *old_cli_buff,
63                   struct user_ia_struct *ia_data )
64 {
65       int debug_on = 1;
66
67       int ok = 0, svr_fd = -1;
68       int num_cli_read = 0, num_svr_read = 0, result = 0;
69       ushort original_priv = 0;
70
71
72       // ptr to attach shared memory to.
73       struct shm_hdr * shm_addr = NULL;
74
75       // get privileges as define in priv_util.c
76       // *********** PRIV CODE ******************/
77       original_priv = enable_priv();
78
79       // set shm_addr to the first addr of the shm structure.
80       if( ss_get_hdr( &(shm_addr), ia_data ) == -1 )
81       {
82          perror("socket_relay: error calling ss_get_hdr");
83          exit(-1);
84       }
85
86       svr_fd = ss_request_connection( shm_addr );
87
88       dbugd(debug_on,"Serving pskt connection: ", svr_fd );
89       dbugd(debug_on," At sl:  ", ia_data->sl );
90       if( svr_fd == -1) {
91          dbug(debug_on, "socket_relay: ss_request_connection
   failed");
92          exit(-1);
93       }
94
95       // move to_cli_buff to shared memory
96       ss_copy_cli_buff( shm_addr, svr_fd, (void*)old_cli_buff );
97
98
99       for ( ;; )
100      {
101         // if there is data to read go get it.
102
103         ok = poll_ok_to_read_block(cli_fd, 50000);
104
105
106         if( ok > 0 )
107         {
108            //dbug(debug_on, "tps_util:server_relay:data avail");
109
110            // data going from tcbe client to protocol svr
```

```
111             num_cli_read = ss_xfer_skt_buff(shm_addr, svr_fd,
    cli_fd );
112
113             // we had a flag indicating there was data to read
114             // if there is actually no data the socket has been
115             // closed. Time to move on.
116             if( num_cli_read <= 0 ) break;
117
118             num_svr_read = ss_xfer_buff_skt(shm_addr, svr_fd,
    cli_fd );
119             if(num_svr_read == -1 ) break;
120
121          } else if(ok < 0 ) {
122
123             if( errno == EINTR ) num_cli_read = 0;
124             else {
125                dbug(debug_on, "Socket no longer
    valid:socket_relay");
126                ss_close( svr_fd, shm_addr );
127                exit(-1);
128             }
129          } else {
130
131             num_cli_read = 0;
132          }// end if
133
134       }// end for loop
135
136       ss_close( svr_fd, shm_addr );
137
138       set_priv( original_priv );
139       //************ END PRIVILEGE CODE ******************/
140       //*************************************************/
141       return result;
142
143 }// end socket_relay
144
145
146 void select_sleep( int fd, long seconds )
147 {
148       int debug_on = 0;
149       static struct timeval timeout;
150
151       fd_set ibits, obits, xbits;
152       FD_ZERO(&ibits);
153       FD_ZERO(&obits);
154       FD_ZERO(&xbits);
155
156       FD_SET(fd, &ibits);
157       FD_SET(fd, &obits);
158       FD_SET(fd, &xbits);
159
160       //timeout.tv_sec = seconds;
161       timeout.tv_sec = 5;
162       timeout.tv_usec = 5;
163
164       if( select(16, &ibits, &obits, &xbits, &timeout ) < 0 )
```

```
165         {
166            dbug( debug_on, "tps_util:select_sleep:select error");
167            perror("select_timer:select error");
168         }
169
170 }// select_timer()
171
```

```
 1 // File:    user_ia.c
 2 // Author:  Scott D. Heller & Susan Bryer-Joyner
 3 // Date:    28 January 1999
 4 // Purpose: Perform User Identification and Authentication.
 5
 6 #include "user_ia.h"
 7
 8 // Have the user enter a user ID, password, session IL and SL
 9 // Ensure these are proper values IAW the user access databases
10 //  used by STOP. This is the procedure that should hook into
11 //  the STOP login procedures.
12 //
13 // Return: user_ia_struct with user_ia_struct.valid set to true if
14 //    valid login data accepted.
15 // *******  This is still a stub **********************
16 // Currently only the valid, sl, and il fields are used.
17 struct user_ia_struct user_IA(int sockfd, struct in_buff_struct
   *queue )
18 {
19    int debug_on = 0;
20
21    const char DELIM = '\n';
22
23    struct user_ia_struct result;
24
25    char  *user_id,
26          *user_pw,
27          *user_sl,
28          *user_il;
29
30    // get info for each of the 4 data items.
31    // much error checking needs to be added here. Currently
32    // anything will be accepted. However the 3rd char of
33    //  user_sl and user_il must be a digit that corresponds
34    //  to the existing level of a protocol server for
35    //  commuication to be established.
36
37
38    user_id = get_token(sockfd, queue, DELIM, MAX_USER_NAME );
39    if( user_id == NULL ) result.valid = false;
40    dbug( debug_on, user_id );
41
42    user_pw = get_token(sockfd, queue, DELIM, MAX_USER_PWD );
43    if( user_pw == NULL ) result.valid = false;
44    dbug( debug_on, user_pw );
45
46    user_sl = get_token(sockfd, queue, DELIM, MAX_SL_LEN );
47    if( user_sl == NULL ) result.valid = false;
48    dbug( debug_on, user_sl );
49
50    user_il = get_token(sockfd, queue, DELIM, MAX_IL_LEN );
51    if( user_il == NULL ) result.valid = false;
52    dbug( debug_on, user_il );
53
54    result.valid = 1;
55    strcpy (result.uname, (const char*)user_id);
56    result.sl = atoi (&user_sl[2]);
```

```
57     result.il = atoi (&user_il[2]);
58
59     if( user_id != NULL ) free(user_id);
60     if( user_pw != NULL ) free(user_pw);
61     if( user_sl != NULL ) free(user_sl);
62     if( user_il != NULL ) free(user_il);
63     dbug( debug_on, "tps_util:user_IA:leaving");
64
65     return result;
66
67 } // end user_IA
68
69
70
```

```
 1 // File: util.c
 2 // Author: Scott D. Heller & Susan Bryer-Joyner
 3 // Date:   28 January 1999
 4 // Purpose: General utility functions possibly used by any
   application
 5
 6 #include "util.h"
 7
 8 // Param:   test value
 9 // Purpose: if test == 0 ensure will print perror information
10 //    and exit.ring 2 application do not have access to assert.
11 void ensure(int test)
12 {
13  int debug_on = 0;
14
15    if(!test){
16       printf("Ensure exiting: %d\n", errno);
17       exit(1);
18    }
19    else{
20       dbug( debug_on, "Ensure ok\n");
21    }//end if
22 } // end ensure
23
24 void ensure_m(int test, char *mssg )
25 {
26     if(!test)
27     {
28       if( mssg != NULL )
29       {
30          printf("Ensure exiting: %s: %d", mssg, errno );
31       } else {
32          printf("Ensure exiting: %d: ", errno );
33       }// end if
34
35   }// end if
36 }// end ensure_m
37
38 // Param:   int on/off switch, debugging message.
39 // Purpose: Standardized debugging. If int is not zero print the
40 //    string prefaced by the pid of the calling process
41 void dbug( int on, char * prompt )
42 {
43    if( on )
44    {
45         if(prompt != NULL )
46         {
47            printf( "%d:%s\n",getpid(), prompt );
48         } else {
49            printf( "NULL" );
50         }
51
52         fflush( stdout );
53    }
54 }
55
56 // Param:   int on/off switch, debugging message.
```

```
57 // Purpose: Standardized debugging. If int is not zero print the
58 //    string prefaced by the pid of the calling process
59 void dbugd( int on, char *prompt, int data)
60 {
61    if(on && prompt != NULL)
62    {
63       printf( "%d:%s %d\n", getpid(), prompt, data );
64         fflush( stdout );
65    }
66 }
67
68
69
```

APPENDIX D. ECHO SERVER SOURCE CODE

Makefile for Echo Server

```
 1 source = echos.c ../listenq.c echo_util.c ../util.c ../io_util.c
   ../buff_io.c ../shm_struct.c ../shm.c ../msem.c ../priv_util.c
 2 CFLAGS = -DUSE_P_SOCKET
 3
 4
 5 echos: ${source}
 6    cc -g -DUSE_P_SOCKET ${source} pskt.c  -o echos -lcass
 7
 8 conf: ${source}
 9    cc -g -DUSE_P_SOCKET ${source} pskt.c  -o echos_c -lcass
10
11 sock: ${source}
12    cc -g -I/usr/include/sys/ ${source} -o echos -lcass -lsocket
13
14 oss: ${source}
15    cc -oss -g -DOSS_OPTION -I/usr/include/sys/ ${source} -o echos
   -lcass -lsocket
16
17 clean:
18    /bin/rm -f /usr2/sdheller/wip/echo/*.o
19    /bin/rm -f /usr2/sdheller/wip/echo/core
20    /bin/rm -f /usr2/sdheller/wip/echo/echos
21
22 depend:
23    cc -Hmake ${CFLAGS} ${source} pskt.c  -o echo -lsocket -lcass
24
25
26
27
28
29
```

```
 1 // File:    echo_util.h
 2 // Author:  Scott D. Heller & Susan Bryer-Joyner
 3 // Date:    28 January 1999
 4 // Purpose: Functions used by the Trusted Path Server (tps.c)
 5
 6
 7 #ifndef TPS_UTIL_H_
 8 #define TPS_UTIL_H_
 9
10 #include <unistd.h>
11 #include <errno.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <memory.h> // for memset()
15
16 //for select in select_sleep
17 #ifndef USE_P_SOCKET
18 #include <sys/time.h>
19 #include <sys/types.h>
20 #include <sys/select.h>
21 #else
22 #include "pskt.h"
23 #endif  // USE_P_SOCKET
24
25
26 #include "../io_util.h"
27 #include "../buff_io.h"
28 #include "../util.h"
29
30 #define MAX_USER_INPUT     256   // longest string accepted from
   user
31 #define MAX_SAK_ATTEMPTS   3     // limit of invalid SAK attempts
   before exit
32 #define MAXHWID            7     // maxsize of hw_id in char + 3
   =>
33                                   //    hw_id can be 3 digits
34 #define TELNET_SEND       255   // value for brk (?)
35 #define TELNET_BRK        243   // value for send (?)
36 #define MIN_SAK_LEN         3    // minumum valid SAK length
37 #define SERV_PORT        6009  // port TPS will listen to.
38
39
40
41
42 // return -1 on error.
43 // relay data from client to server and vice a versa.
44 int socket_relay(int cli_fd, struct in_buff_struct *cli_buff );
45
46 // select test
47 void select_sleep(int, long );
48
49
50 #endif
```

```
1  // File:    pskt.h
2  // Author:  Scott Heller
3  // Date:    20 February 1999
4  // Purpose: Provide socket like interface to shared memory
   information
5  //    passing.
6
7  #ifndef PSKT_H_
8  #define PSKT_H_
9
10 // decl needed to simulate socket.h
11
12 #define I_NREAD 1       // this means ioctl needed
13 #define AF_INET -1      // only socket type supported internet
   stream.
14 #define SOCK_STREAM -1
15
16 #define MAX_OPEN_CONN 5 // sets number of connection buffers
   allocated in shm.
17
18 /*
19  * defined in shm_struct.h
20  * struct sockaddr {
21  *    u_char sa_len;
22  *    u_char sa_family;
23  *    char   sa_data[14];
24  * };
25  */
26
27 struct timeval {
28    int tv_sec;
29    int tv_usec;
30 };
31
32 typedef int fd_set[MAX_OPEN_CONN]; // for select
33
34 #include "../msem.h"
35 #include "../shm.h"
36 #include "../listenq.h"
37 #include "../shm_struct.h"
38
39
40
41 // Return:  pseudo-socket descriptor for listen queue.
42 //    Fixed at MAX_OPEN_CONN + 1.
43 // Param:   domain: Not used. Should expect AF_INET
44 //          type:   Not used. Should be SOCK_STREAM
45 //          protocol: Not used.
46 // Purpose: Provide pseudo-socket interface consitant with tcp/ip
   sockets.
47 //    Initializes shared memory structure used to simiulate socket
   connections.
48 int socket(int domain, int type, int protocol );
49
50
51 // Return:  As expected for socket bind. 0 on success, -1
   otherwise.
```

```
52 // Param:   sockfd:  Must be listen queue socket descriptor.
53 //          sockaddr: Not used.
54 //          size:    Not used.
55 // Purpose: Provide pseudo-socket interface consitant with tcp/ip
   sockets.
56 int bind(int sockfd, const struct sockaddr * serv_addr, int size
   );
57
58 // Return:  As expected for socket bind. 0 on success, -1
   otherwise.
59 // Param:   fd:   Must be listen queue socket descriptor.
60 //          queue_size: Not used. Hard coded to 5 during the
   socket call.
61 //    Future work should cause the allocation of shm to be delayed
   until
62 //    here. Then queue_size could drive the size of the shm
   segment.
63 //    Requires modification to shm_struct initialization procedure
   and
64 //    struct declarations.
65 // Purpose: Provide pseudo-socket interface consitant with tcp/ip
   sockets.
66 int listen( int fd, int queue_size );
67
68
69 // Return:  Pseudo-socket identifier. (AKA connection index).
70 // Param:   listen_sem: the listen queue identifier from the call
   to socket.
71 //          addr: Not used. Future should get actual client
   address from SSS
72 //          addr_len: Not used. Future should return len of addr.
73 // Purpose: Provide pseudo-socket interface consitant with tcp/ip
   sockets.
74 int accept(int listen_sem, struct sockaddr * addr, int * addr_len
   );
75
76
77 // Return:  number of char read. -1 on error.
78 // Param:   fd: pseudo-socket connection id.
79 //          buff: location for char data.
80 //          read_limit: max char to read.
81 // Purpose: Provide pseudo-socket interface consitant with tcp/ip
   sockets.
82 int my_read(int fd, char *buff, int read_limit);
83
84
85 // Param:   fd: pseudo-socket connection id.
86 // Purpose: Provide pseudo-socket interface consitant with tcp/ip
   sockets.
87 void my_close( int fd );
88
89
90 // Return:  number of char written. -1 on error.
91 // Param:   fd: pseudo-socket connection id.
92 //          buff: location for char data to write.
93 //          nbytes: max char to write.
```

```
94 // Purpose: Provide pseudo-socket interface consitant with tcp/ip
   sockets.
95 int my_write(int fd, const char* data, int nbytes );
96
97
98 // Return:  number of pseudo-socket descriptors with bits set.
99 // Param:   bits_to_check: Not used. MAX_OPEN_CONN is hard coded.
100 //          ibits:   set of bits. each sckt id set using FD_SET is
    checked
101 //             for data availible. Set to 1 if data avail upon
    select return.
102 //          obits:   set of bits. each sckt id set using FD_SET is
    checked
103 //             for space availible. Set to 1 if space avail upon
    select return.
104 //          xbits:   set of bits. each sckt id set using FD_SET is
    checked
105 //             for connection valid. Set to 1 if connection
    invalid upon
106 //             select return.
107 //          timeout: Not currently used. Should indicate max
    blocking time.
108 // Purpose: Provide pseudo-socket interface consitant with tcp/ip
    sockets.
109 int select( int bits_to_check, fd_set  *ibits, fd_set *obits,
    fd_set *xbits,
110                struct timeval *timeout );
111
112
113 // Param:   fd: pseudo-socket connection id.
114 //          bits: set of flag bits of which one should be
    associated with fd.
115 // Purpose: Provide pseudo-socket interface consitant with tcp/ip
    sockets.
116 void FD_SET(int fd, fd_set *bits);
117
118
119 // Param:   bits: set of flag bits to be set to all ZERO.
120 // Purpose: Provide pseudo-socket interface consitant with tcp/ip
    sockets.
121 void FD_ZERO(fd_set *bits);
122
123
124 // Return:  true if set. false otherwise.
125 // Param:   fd: pseudo-socket connection id.
126 //          bits: set of flag bits to test if fd is set to
    true(1).
127 // Purpose: Provide pseudo-socket interface consitant with tcp/ip
    sockets.
128 int  FD_ISSET(int fd, fd_set *bits);
129
130
131 // Param:   fd: pseudo-socket connection id.
132 //          bits: set of flag bits of which the bit associated
    with fd will be
133 //             set to zero.
```

```
134 // Purpose: Provide pseudo-socket interface consitant with tcp/ip
    sockets.
135 void  FD_CLEAR(int fd, fd_set *bits);
136
137
138 //ioctrl - not yet designed.
139 //fctrl - not yet designed.
140
141 #endif
```

```
 1 // File: echo_util.c
 2 // Author: Scott D. Heller & Susan Bryer-Joyner
 3 // Date:   28 January 1999
 4 // Purpose: Functions used by the Trusted Path Server (tps.c)
 5
 6 #include "echo_util.h"
 7
 8
 9 int socket_relay( int cli_fd, struct in_buff_struct *cli_buff )
10 {
11       int debug_on = 0;
12       dbug(debug_on, "tps_util:socket_relay:entered");
13
14       int to_svr_nbytes = 0, ok = 0;
15       int num_cli_read = 0, num_svr_read = 0, result = 0;
16       int lq_shmid = 0;
17       char *to_svr;
18
19       //select test ************
20       fd_set ibits;
21       fd_set obits;
22       fd_set xbits;
23       FD_ZERO(&ibits);
24       FD_ZERO(&obits);
25       FD_ZERO(&xbits);
26
27       //timeout.tv_sec = seconds;
28       static struct timeval timeout;
29       timeout.tv_sec = 5;
30       timeout.tv_usec = 5;
31
32       dbugd( debug_on, "cli_fd = ", cli_fd );
33
34       for ( ;; )
35       {
36          // if there is data to read go get it.
37          dbug(debug_on, "******about to call FD_SET ********");
38
39           FD_SET(cli_fd, &ibits);
40 //        FD_SET(cli_fd, &obits);
41           FD_SET(cli_fd, &xbits);
42           dbug(debug_on, "******about to call select ********");
43           if( select(16, &ibits, &obits, &xbits, &timeout ) < 0 )
44           {
45            dbug( debug_on, "tps_util:select_sleep:select error");
46            perror("select_timer:select error");
47           }
48           dbugd( debug_on, "ibits = ", FD_ISSET(cli_fd, &ibits) );
49           dbugd( debug_on, "obits = ", FD_ISSET(cli_fd, &obits) );
50           dbugd( debug_on, "xbits = ", FD_ISSET(cli_fd, &xbits) );
51
52 //       ok = poll_ok_to_read(cli_fd);
53        ok = FD_ISSET(cli_fd, &ibits);
54          dbugd( debug_on, "echo_util:session_server: FD_ISSET(
   ibits ) =",
55              ok );
56
```

```
57          if( ok > 0 )
58          {
59             dbug(debug_on, "tps_util:server_relay:data avail");
60             num_cli_read = get_data( cli_fd, cli_buff );
61             dbugd( debug_on, "echo:get_data read = ", num_cli_read
   );
62             if(debug_on) print_buff_queue( cli_buff );
63
64             // we had a flag indicating there was data to read
65             // if there is actually no data the socket has been
66             // closed. Time to move on.
67             if( num_cli_read <= 0 ) break;
68
69             if(debug_on) print_buff_queue( cli_buff );
70
71             to_svr_nbytes = num_char( cli_buff );
72             to_svr = empty_buff( cli_buff);
73             dbugd( debug_on, "echos: There are nbytes in my buff
   => ",
74                to_svr_nbytes );
75             dbug( debug_on, "echos: writing the following to
   Writen...");
76             dbug( debug_on, to_svr );
77 #ifdef DEMO
78             dbug( 1, to_svr );
79 #endif //DEMO
80
81             if( to_svr != NULL )
82             {
83                if( Writen( cli_fd, to_svr, to_svr_nbytes ) < 0 )
84                {
85                   perror( "relay: Writen error");
86                   exit(-1);
87
88                }// end if
89                free(to_svr);
90             }// end if
91
92          } else if(ok == 0 && FD_ISSET( cli_fd, &xbits) ) {
93
94             perror("Socket no longer valid:socket_relay");
95             my_close( cli_fd );
96             exit(-1);
97          } else {
98
99             num_cli_read = 0;
100         }// end if
101
102      }// end for loop
103
104      return result;
105
106 }// end socket_relay
107
108
109 void select_sleep( int fd, long seconds )
110 {
```

```
111        int debug_on = 0;
112        static struct timeval timeout;
113
114        fd_set ibits, obits, xbits;
115        FD_ZERO(&ibits);
116        FD_ZERO(&obits);
117        FD_ZERO(&xbits);
118
119        FD_SET(fd, &ibits);
120        FD_SET(fd, &obits);
121        FD_SET(fd, &xbits);
122
123        //timeout.tv_sec = seconds;
124        timeout.tv_sec = 5;
125        timeout.tv_usec = 5;
126
127        if( select(16, &ibits, &obits, &xbits, &timeout ) < 0 )
128        {
129           dbug( debug_on, "tps_util:select_sleep:select error");
130           perror("select_timer:select error");
131        }
132
133 }// select_timer()
134
```

```
 1 // File: echo.c
 2 // Author: Scott D. Heller & Susan Bryer-Joyner
 3 // Date:   28 January 1999
 4 // Purpose: Main() for the Trusted Path Server
 5
 6
 7 #include    <errno.h>
 8 #include    <stdio.h>            //
 9 #ifndef USE_P_SOCKET
10 #include    <types.h>
11 #include    <sys/socket.h>
12 #else
13 #include "pskt.h"               // inplace of sys/socket.h
14 #endif //USE_P_SOCKET
15
16 #ifdef OSS_OPTION
17 #include    <stop/tcb_gates.h>   // used for fork_process
18 #include    <error_code.h>       // for suspend event
19 #include    <message.h>          // for suspend event
20 #else
21 extern int fork();
22 #endif //OSS_OPTION
23
24 #include    <netinet/in.h>
25 #include    <string.h>
26 #include    <unistd.h>
27 #include    <stdlib.h>
28 #include    <sys/byteorder.h>    // for htonl and htons
29
30 #include    <fcntl.h>
31
32 #include    "../util.h"
33 #include    "echo_util.h"
34 #include    "../cdb.h"
35 #include   "../buff_io.h"
36
37 extern int fcntl( int, int, int );
38
39 #ifdef OSS_OPTION
40 #define sleep(a)
   suspend_event(NO_EVENT,(a)*ONE_SECOND,0,NULL,NULL,NULL)
41 #define fork() fork_process()
42 #endif
43
44 int main ()
45 {
46       printf("hello\n");
47    int   debug_on = 1;
48
49    int   listenfd = 0,
50          connfd = 0,
51          clilen = 0,
52          testBind = 0,
53            flag   = 0;
54
55
56      struct in_buff_struct *buffer;
```

```
57
58    dbugd(debug_on, "Start execution.TPS pid = ", getpid() );
59
60    struct   sockaddr_in cliaddr, servaddr;
61
62    listenfd = socket(AF_INET, SOCK_STREAM, 0);
63
64    ensure( listenfd > -1 );
65
66    memset( &servaddr, 0, sizeof(servaddr) );
67
68    servaddr.sin_family      = AF_INET;
69    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
70    servaddr.sin_port        = htons(SERV_PORT);
71
72    testBind = bind(listenfd,(struct
   sockaddr*)&servaddr,sizeof(servaddr));
73    ensure( testBind > -1 );
74
75    printf("Listening to port %d\n", SERV_PORT);
76    listen( listenfd, 5 );
77
78    for ( ; ; )
79    {
80       clilen = sizeof( cliaddr );
81
82       // block until connection then accept
83       connfd = accept( listenfd, (struct sockaddr *) &cliaddr,
   &clilen );
84         dbugd( debug_on, "echos:returned from accept connfd = ",
   connfd );
85
86       ensure (connfd > -1);
87
88       if( ( fork() )  == 0 )
89       {
90          // child process
91          dbugd( debug_on, "TPS Child. pid = ", getpid() );
92
93          close(listenfd);
94            dbug( debug_on, "TPS Child: calling malloc struct
   in_buff_struct");
95            buffer = malloc( sizeof( struct in_buff_struct ) );
96            dbug( debug_on, "TPS Child: calling
   init_buffer(buffer)");
97            init_buffer( buffer );
98
99            dbug(debug_on, "tps:calling socket_relay");
100           socket_relay( connfd, buffer );
101
102           free( buffer );
103
104           dbugd(debug_on, "Exiting!! :", getpid() );
105        exit(0);
106      }//end if
107
108      close(connfd); // parent closes connected socket
```

```
109
110     }// end for loop
111
112 }// end main
113
```

```
1 // File:    pskt.c
2 // Author:  Scott Heller
3 // Date:    20 February 1999
4 // Purpose: Provide socket like interface to shared memory
  information
5 //    passing.
6
7
8 #include "pskt.h"
9
10 static struct shm_hdr *shmhdr;   // required since protcol svr can
  not pass.
11 static int initialize = 1;       // ensure we only initalize once.
12 static int child_needs_shm = 1;  // ensure we attach if needed.
13
14
15 // initialize static shm database if needed otherwise return.
16 int socket(int domain, int type, int protocol )
17 {
18    int debug_on = 1;
19    key_t level_key = 0;
20    int shmid = 0;
21
22    if( initialize )
23    {
24       dbug(debug_on, "socket:Initializing shm_hdr");
25
26       // initialize shared memory.
27       shmid = init_shm_hdr( &shmhdr );
28       if( shmid < 0 ) {
29          perror("init_shm_hdr: failed");
30          exit(-1);
31       }
32
33       // prove able to access memory
34       dbugd( debug_on, "If able to read mem access worked here",
35                shmhdr->conn[0].in_use );
36       initialize = 0;
37    } else {
38       dbugd( debug_on, "socket: called and NOT initializing",
39             initialize );
40    }// end if
41
42    // fixed value for listen queue p-socket.
43    return MAX_OPEN_CONN + 1;
44
45 } // end socket()
46
47
48 //bind
49 int bind(int sockfd, const struct sockaddr * serv_addr, int size )
50 {
51    int debug_on = 1;
52    int result = 0;   // success
53    // do nothing
54
55    if(sockfd != MAX_OPEN_CONN + 1) {
```

```
56        dbugd(debug_on, "bind:server bind to unexpected sockfd = ",
   sockfd);
57        result = -1;
58        ss_cleanup(shmhdr);
59     }
60
61     return result;
62  }// end bind()
63
64
65  //listen
66  int listen( int fd, int queue_size )
67  {
68     // do nothing
69     int debug_on = 1;
70     int result = 0;// success
71
72     if( fd != MAX_OPEN_CONN + 1) {
73        dbugd(debug_on, "listen:server listen to unexpected sockfd =
   ", fd );
74        ss_cleanup(shmhdr);
75        result = -1;
76     }
77
78     return result;
79  }// end listen()
80
81
82  //accept
83  int accept(int listen_fd, struct sockaddr * addr, int * addr_len )
84  {
85     int   debug_on    = 1;
86
87     int   new_skt_id  = -1;
88
89     struct listen_q_struct *lq = NULL;
90
91     // block until connection availible
92     new_skt_id = ss_block_on_lq( shmhdr );
93
94     dbugd( debug_on, "accept:new_skt_id = ", new_skt_id );
95
96     // return connection index
97     return new_skt_id;
98
99  }// end accept
100
101 int select( int bits_to_check, fd_set *tibits, fd_set *tobits,
   fd_set *txbits,
102              struct timeval *timeout )
103 {
104       int debug_on = 0;
105       dbug(debug_on, "select: entered");
106       int *ibits, *obits,* xbits;
107       ibits = *tibits;
108       obits = *tobits;
109       xbits = *txbits;
```

```
110       int result = 0, set_one = 0;
111       int shmid = 0;
112
113       if( child_needs_shm )
114       {
115          shmid = ss_get_hdr( &shmhdr, NULL );
116          child_needs_shm = 0;
117       }
118
119
120       sleep( 1 ); // psuedo block should be fixed via signal io.
121
122       for(int idx = 0; idx < MAX_OPEN_CONN; idx++)
123       {
124          if(debug_on && shmhdr->conn[idx].in_use)
125          {
126             dbugd( debug_on, "select: conn in_use => ", idx );
127          }
128
129          set_one = 0;
130          // determine if we should set a bit
131          // if a bit set for a fd (aka idx) increment result
132          // it is reasonable that a bit set will be null. Must
    check for this.
133          if(ibits != NULL && ibits[idx])
134          {
135             if( (ibits[idx] = ss_data_avail(idx, shmhdr)) > 0 )
136             {
137                dbugd( debug_on, "select:ibit set for fd = ", idx
    );
138                set_one = 1;
139                result++;
140             }
141          }
142
143          if(obits != NULL && obits[idx])
144          {
145             obits[idx] = ss_space_avail(idx, shmhdr);
146             if( obits[idx] ) dbugd( debug_on, "select:obit set for
    fd = ", idx );
147             if(!set_one && obits[idx])
148             {
149                result++;
150                set_one++;
151             }
152          }
153
154          if(xbits != NULL && xbits[idx])
155          {
156             xbits[idx] = ss_socket_error(idx, shmhdr);
157             if( xbits[idx]) dbugd(debug_on,"select:should be NOT
    in_use",
158                shmhdr->conn[idx].in_use );
159             if( xbits[idx] ) dbugd( debug_on, "select:xbit set for
    fd = ", idx );
160             if(!set_one && xbits[idx])
161             {
```

```
162                 result++;
163                 set_one++;
164              }
165           }
166        }
167
168        dbugd( debug_on, "select exiting: result = ", result );
169        return result;
170
171 }// end select()
172
173 void FD_ZERO( fd_set *tbits)
174 {
175        int * bits = *tbits;
176
177        for(int idx = 0; idx < MAX_OPEN_CONN; idx++ )
178        {
179           bits[idx] = 0;
180        }
181 }// end FD_ZERO()
182
183
184 void FD_SET(int fd, fd_set *tbits)
185 {
186        ensure_m( fd >= 0 && fd < MAX_OPEN_CONN, "FD_SET: invalid
    fd" );
187        int *bits = *tbits;
188        bits[fd] = 1;
189 }// end FD_SET()
190
191 int FD_ISSET(int fd, fd_set* tbits)
192 {
193        ensure_m( fd >= 0 && fd < MAX_OPEN_CONN, "FD_ISSET: invalid
    fd" );
194        int *bits = *tbits;
195        return (bits[fd] > 0 ? 1 : 0);
196 }//end FD_ISSET
197
198 void FD_CLEAR(int fd, fd_set* tbits)
199 {
200        ensure_m( fd >= 0 && fd < MAX_OPEN_CONN, "FD_CLEAR: invalid
    fd" );
201        int *bits = *tbits;
202        bits[fd] = 0;
203 }// end FD_CLEAR
204
205
206 //myread
207 int my_read(int fd, char *buff, int read_limit)
208 {
209    int n = 0;
210
211    n = ss_read(fd, shmhdr, buff, read_limit);
212
213    return n;
214 }// end my_read
215
```

```
216
217 //mywrite
218 int my_write(int fd, const char* data, int nbytes )
219 {
220    int n = 0;
221
222    n = ss_write( fd, shmhdr, data, nbytes );
223
224    return n;
225 }// end my_write
226
227 void my_close( int fd )
228 {
229       int debug_on = 1;
230
231       ss_close( fd, shmhdr );
232
233 }// end my_close()
234
235
236
237 //ioctrl
238 //fctrl
239
240
241
```

APPENDIX E. PSUEDO-TCBE SOURCE CODE

Makefile for Pseudo-Trusted Computing Base Extension

```
1 source = tcbe.c ../util.c  ../io_util.c ../cdb.c ../buff_io.c
  cli_echo.c
2
3 tcpserv: ${source}
4    cc -g -I/usr/include/sys/ ${source} -o tcbe -lsocket -lcass
5
6 clean:
7    /bin/rm -f /usr2/sdheller/wip/tcbe/*.o
8    /bin/rm -f /usr2/sdheller/wip/tcbe/core
9
10
11
```

```
 1 // File: cli_echo.h
 2 // Author: Scott Heller and Susan BryerJoyner
 3 // Date:   2 Feb 1999
 4 // Purpose: echo client function
 5
 6 #include <stdio.h>
 7 #include <string.h>
 8
 9 #include "../io_util.h"
10 #include "../buff_io.h"
11
12 #ifndef CLI_ECHO_H_
13 #define CLI_ECHO_H_
14
15 #define MAXLINE 4096
16
17 // perform the echo client funtions.
18 void str_cli( int sockfd );
19
20
21 #endif
22
23
```

```
 1 // File: cli_echo.c
 2 // Author: Scott Heller and Susan BryerJoyner
 3 // Date:   2 Feb 1999
 4 // Purpose: echo client function
 5
 6 #include "../util.h"
 7 #include "cli_echo.h"
 8
 9
10 void str_cli( sockfd )
11 register int sockfd;
12 {
13       int debug_on = 1;
14       int n, ok = 0;
15       char *recvline, sendline[MAXLINE];
16       struct in_buff_struct *recvline_ptr = malloc(sizeof (struct
   in_buff_struct));
17       dbug(debug_on, "Entered str_cli: Echo server should be
   responding.");
18
19       while (fgets(sendline, MAXLINE, stdin) != NULL)
20       {
21
22             n = strlen(sendline);
23             dbug( debug_on, "attempting to Writen");
24             if (n > 1)
25             {
26                dbug( debug_on, "calling Writen");
27                if (Writen(sockfd, sendline, n) < 0)
28                {
29                   break;
30                }
31                else
32                {
33                dbugd( debug_on, "wrote: ", n);
34                }//end if
35             }//end if
36
37          ok = poll_ok_to_read( sockfd );
38          // now read a line from the socket and write it to
39          // our standard output
40          if( ok == 1 )
41          {
42             dbug( debug_on, "attmpting to Readline");
43             if( (n = get_data(sockfd, recvline_ptr)) > 0 )
44             {
45                // empty_buff allocates memory for and rtns char *.
46                // must free(recvline) after done using.
47                recvline = empty_buff( recvline_ptr );
48
49                dbugd( debug_on, "Readline read n bytes => ", n);
50
51                if( recvline != NULL )
52                {
53                   puts(recvline );
54                   free( recvline );
55                }
```

```
56                if(n == 0) break;  // socket was closed.
57             } else if( n < 0 )
58             {
59                break;
60             }// end if
61
62          } else if( ok < 0 ) break; //end if
63       }// end for
64       free( recvline_ptr );
65
66       dbug( debug_on, "sti_cli - leaving" );
67 }// end str_cli
```

```
   // File: tcbe.c

 1 #include  <sys/socket.h>
 2 #include  <netinet/in.h>
 3 #include  <sys/byteorder.h>
 4 #include  <arpa/inet.h>
 5 #include  <fcntl.h>
 6
 7 #include  "../util.h"
 8 #include  "../io_util.h"
 9 #include  "../buff_io.h"
10 #include  "cli_echo.h"
11
12 #define MAX_USER_INPUT 256
13
14 extern int fcntl( int, int, int );
15
16 int
17 main( int argc, char* argv[] )
18 {
19    int debug_on = 1;
20
21    int                  sockfd;
22    int                  flags = 0;
23    struct sockaddr_in   servaddr;
24
25    int cnt = 0; // counts number of elements in SAK
26
27    char SAK[20];
28    //SAS beginning.
29    SAK[cnt++] = (unsigned int)255;
30    SAK[cnt++] = (unsigned int)243;
31
32    //add user entered hardware id.
33    while( (*argv[1]) != '\0' )
34    {
35          SAK[cnt++] = *(argv[1]++);
36    }
37
38    // add delimiter to SAS
39    SAK[cnt++]  = '\n';
40    SAK[cnt++]  = '\0';
41    SAK[cnt]    = '\0';
42
43
44    if (debug_on)
45    {
46       printf ("cnt: %d\n", cnt);
47       cnt = 0;
48       while(SAK[cnt] != '\0')
49       {
50          putchar(SAK[cnt++]);
51       }//end while
52    }//end if
53
54    //interactive input from keyboard
55    char user_IA_mssg[MAX_USER_INPUT];
```

```
56    char user_pwd[20];
57    char user_sl[20];
58    char user_il[20];
59
60    printf("Enter user name\n?");
61    scanf ("%s", user_IA_mssg);
62
63    printf("Enter password\n?");
64    scanf ("%s", user_pwd);
65
66    strcat(user_IA_mssg, "\n");
67    strcat(user_IA_mssg, user_pwd);
68
69    printf("Enter new security level\n?");
70    scanf ("%s", user_sl);
71
72    strcat(user_IA_mssg, "\n");
73    strcat(user_IA_mssg, user_sl);
74
75    printf("Enter new integrity level\n?");
76    scanf ("%s", user_il);
77
78    strcat(user_IA_mssg, "\n");
79    strcat(user_IA_mssg, user_il);
80    strcat(user_IA_mssg, "\n");
81
82    printf("%s", user_IA_mssg);
83    memset(&servaddr, 0, sizeof(servaddr));
84    servaddr.sin_family      = AF_INET;
85    servaddr.sin_addr.s_addr = inet_addr("131.120.10.99");
86    servaddr.sin_port        = htons(6002);
87
88    dbug( debug_on, "entering tcbe" );
89    if( (sockfd = socket(AF_INET, SOCK_STREAM, 0 )) < 0 )
90       perror("client socket call failed");
91
92    dbugd( debug_on, "sockfd = ", (int)sockfd );
93
94    if( (connect(sockfd, (struct sockaddr *) &servaddr,
   sizeof(servaddr)) ) < 0 )
95    {
96       perror("client cannot connect to server");
97       exit(-1);
98    }
99
100   flags = fcntl( sockfd, F_GETFL, 0);
101   //  flags |= O_NDELAY;
102
103   if( fcntl( sockfd, F_SETFL, flags ) == -1 )
104   {
105      perror("fcntl failed");
106      exit(-1);
107   }
108   if( debug_on )
109   {
110      flags = fcntl( sockfd, F_GETFL, 0);
111      if( flags & O_NDELAY ) dbug( debug_on, "O_NDELAY SET");
```

```
112         else dbug( debug_on, "O_NDELAY OFF");
113     }
114     dbugd( debug_on, "tcbe:strlen(SAK) = ", strlen(SAK) );
115     dbug( debug_on, SAK );
116     Writen( sockfd, SAK, strlen(SAK) );
117     dbug( debug_on, user_IA_mssg);
118     Writen( sockfd, user_IA_mssg, strlen(user_IA_mssg) );
119     dbug( debug_on, "tcbe: finished writing user_IA_mssg");
120
121     str_cli(sockfd);      /* do it all */
122     sleep(2);
123     exit(0);
124 }
```

APPENDIX F. GLOSSARY OF TERMS AND ACRONYMS

*-Property – A Bell-LaPadula security model rule allowing a subject write access to an object only if the security level of the subject is dominated by the security level of the object. [Ref. 2]

Access – A specific type of interaction between a subject and an object that results in the flow of information from one to the other. [Ref. 2]

Access Control – (1) The limiting of rights or capabilities of a subject to communicated with other subjects, or to use functions or services in a computer system or network. (2) Restrictions controlling a subject's access to an object. [Ref. 7]

Accountability – Accountability is the quality or state that enables actions on an ADP system to be traced to individuals who may then be held responsible. These actions include violations and attempted violations of the security policy, as well as allowed actions.

Audit Trail – A set of records that collectively provide documentary evidence of processing used to aid in tracing from original transactions forward to related records and reports, and/or backwards from records and reports to their component source transactions. [Ref. 7]

Authentication – (1) To establish the validity of a claimed identity. (2) To provide protection against fraudulent transactions by establishing the validity of message, station, individual, or originator. [Ref. 7]

Bell-LaPadula Model – A formal state transition model of computer security policy that describes a set of access control rules. In this formal model, the entities in a computer system are divided into abstract sets of subjects and objects. The notion of a secure state is defined and it is proven that each state transition preserves security by moving from secure state to secure state; thus, inductively proving that the system is secure. A system state is defined to be "secure" if the only permitted access modes of subjects to objects are in accordance with a specific security policy. In order to determine

whether or not a specific access mode is allowed, the clearance of a subject is compared to the classification of the object and a determination is made as to whether the subject is authorized for the specific access mode. The clearance/classification scheme is expressed in terms of a lattice. [Ref. 2]

Daemon – A daemon is a process that runs in the background and is independent of control from all terminals. [Ref. 20]

Denial of Service – The prevention of authorized access to system assets or services, or the delaying of time critical operations. [Ref. 7]

Diffusion – A method in which the statistical structure of the plain text is dissipated into long-range statistics of the cipher text. This is achieved by having each plain text digit affect the value of many cipher text digits. [Ref. 9]

Discretionary Access Control (DAC) – A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control). [Ref. 2]

Dominate – Security level S_1 is said to dominate security level S_2 if the hierarchical classification of S_1 is greater than or equal to that of S_2 and the non-hierarchical categories of S_1 include all those of S_2 as a subset. [Ref. 2]

Lattice – A partially ordered set for which every pair of elements has a greatest lower bound and a least upper bound. [Ref. 2]

Module – In software, a module is part of a program. Programs are composed of one or more independently developed modules that are not combined until the program is linked. A single module can contain one or several routines. [Ref. 30]

Nonce – A unique character string used in cryptography to provide protection against replay attacks. [Ref. 10]

Object – A passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. [Ref. 7]

Penetration – The successful violation of a protected system. [Ref. 7]

Process – A program in execution. It is completely characterized by a single current execution point (represented by machine state) and address space. [Ref. 7]

Reliability – The extent to which a system can be expected to perform its intended function with required precision. [Ref. 7]

Secure Attention Sequence (SAS) – An out-of-band communication from a trusted computing base extension (TCBE) to either the Trusted Path Server (TPS) or the Session Server associated with an active session. It is composed of the TCBE hardware identification number and a nonce and encrypted using the Secure LAN Server public-key.

Secure Local Area Network Server – A software product composed of a Trusted Path Server and a Session Server that provides a method of establishing a secure session over a trusted path. The Trusted Path Server establishes the trusted path between the trusted computing base (TCB) of a high assurance server and a TCB extension (TCBE) over an Ethernet Local Area Network. The Session Server receives the information required to establish a secure session via the trusted path. The Session Server then provides an interface to ported protocol servers.

Security Policy – The set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information. [Ref. 7]

Session Server – The Session Server is a software component of the Secure LAN Server. It provides the hardware and user authentication required to establish the trusted path and the secure session, respectively. Upon successful authentication, it provides a relay between the TCBE and the ported protocol servers.

Simple Security Property – A Bell-LaPadula security model rule allowing a subject read access to an object only if the security level of the subject dominates the security level of the object. [Ref. 2]

Subject – An active entity, generally in the form of a person, process, or device that causes information to flow among objects or changes the system state. Technically, a process/domain pair. [Ref. 7]

Trusted Computing Base (TCB) – The totality of protection mechanisms within a computer system – including hardware, firmware, and software – the combination of which is responsible for enforcing a security policy. It creates a basic protection environment and provides additional user services required for a trusted computer system. The ability of a trusted computing base to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters related to the security policy. [Ref. 7]

Trusted Computing Base Extension (TCBE) – A network interface card (NIC) that has been modified to support a trusted path to the trusted computing base (TCB) on the XTS-300.

Trusted Path Server (TPS) – A program that initializes the Connection Database and handles requests for new connections.

Trusted Subject – (1) A trusted subject is a subject that is part of the TCB. It has the ability to violate the security policy, but is trusted not to actually do so. [Ref. 7] (2) In the XTS-300, a trusted subject is one that has an integrity level that allows manipulation of TCB databases (an integrity level of at least operator) or if the process possess privileges that exempt it from specific access control rules (for example, the privilege to be exempt from the security *-property). [Ref. 18]

LIST OF REFERENCES

1. *A Guide to Understanding Security Modeling in Trusted Systems*, NCSC-TG-010 Version 1, National Computer Security Center, October 1992.

2. *Department of Defense Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, National Computer Security Center, December 1985.

3. *Navy Virtual Intranet: Functional Architecture and Concept of Operations*, Draft Version, Naval Virtual Intranet Integrated Process Team, 11 December 1997.

4. Irvine, C.E., Anderson, J.P., Robb, D.A., and Hackerson, J., "High Assurance Multilevel Services for Off-The-Shelf Workstation Applications", *Proceedings of the National Information Systems Security Conference*, October 1998.

5. Cisco Systems, [http://www.cisco.com/public/library/isakmp/isakmp.html].

6. Whittle, R., "Cryptography for Encryption, Digital Signatures and Authentication", [http://www.ozemail.com.au/~firstpr/crypto/index.html].

7. *Trusted Network Interpretation of The Trusted Computer System Evaluation Criteria*, NCSC-TG-005 Version-1, National Computer Security Center, 31 July 1987.

8. National Security Agency Evaluated Products List, [http://www.radium.ncsc.mil/tpep/epl/entries/CSC-EPL-92-003-D.html].

9. Stallings, W., *Cryptography and Network Security: Principles and Practice*, 2nd Edition, Prentice Hall, Inc., 1998.

10. Garfinkel, S. and Spafford, G., *Practical UNIX and Internet Security*, 2nd Edition, O'Reilly & Associates, Inc., 1996.

11. *DoD Information Security Program*, Department of Defense Directive 5200.1-R, January 1997.

12. *Glossary of Computer Security Terms*, National Computer Security Center, 21 October 1988.

13. *An Introduction to Computer Security: The NIST Handbook*, NIST Special Publication 800-12, National Institute of Standards and Technology, October 1995.

14. *Department of Defense Password Management Guideline*, CSC-STD-002-85, Department of Defense Computer Security Center, 12 April 1985.

15. *Security Requirements for Automated Information Systems (AISs)*, Department of Defense Directive 5200.28, March 21, 1988.

16. "Concepts and Terminology for Computer Security," *Information Security: An Integrated Collection of Essays*, IEEE Computer Society Press, 1995.

17. Garfinkel, S. and Spafford, G., *Practical UNIX & Internet Security*, O'Reilly & Associates, Inc., 1996.

18. *XTS-300, Trusted Facility Manual*, Document ID: FS92-371-07, WANG Government Services, Inc., McLean, VA, March 1998.

19. Parnas, D.L., "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, December 1972.

20. Stevens, W. R., *UNIX Network Programming Volume 1, Networking APIs: Sockets and XTI*, 2nd Edition, Prentice Hall, Inc., 1998.

21. *XTS-300, User's Manual*, Document ID: FS92-373-07, Wang Government Services, Inc., McLean, VA, March 1998.

22. Stevens, W. R., *UNIX Network Programming*, Prentice Hall, Inc., 1990.

23. Kang, M.H., Moore, A. and Moskowitz, I.S, "Design and Assurance Strategy for the NRL Pump," *Computer*, Institute of Electrical and Electronics Engineers, Inc., April 1998.

24. Kang, M.H., Froscher, J.N. and Eppinger, B.J., "Towards an Infrastructure for MLS Distributed Computing," *Proceedings of the Computer Security Applications Conference*, Institute of Electrical and Electronics Engineers, Inc., April 1998.

25. Eads, B. *Developing a High Assurance Multilevel Mail Server*, Naval Postgraduate School, Monterey, CA, March 1999.

26. *XTS-300, STOP 4.4, Network Login Option Software Release Bulletin*, Document ID: FB94-237-04, WANG Federal, Inc., McLean, VA, April 1997.

27. *XTS-300, Pentium Installation and Setup*, Document ID: FS96-290-04, WANG Government Services, Inc., McLean, VA, March 1998.

28. *XTS-300, STOP 4.4, Trusted Programmer's Reference Manual*, Document ID: FS92-375-07, WANG Government Services, Inc., McLean, VA, March 1998.

29. *XTS-300, STOP 4.4, Application Programmer's Reference Manual*, Document ID: FS92-374-06, WANG Government Services, Inc., McLean, VA, March 1998.

30. PC Webopedia, [http://webopedia.internet.com].

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Chairman, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Dr. Cynthia E. Irvine 3
   Computer Science Department Code CS/Ic
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. Mr. James Bret Michael 1
   Computer Science Department Code CS/Mb
   Naval Postgraduate School
   Monterey, CA 93943-5000

6. Mr. James P. Anderson 1
   James P. Anderson Company
   Box 42
   Fort Washington, PA 19034

7. Mr. Joseph O'Kane 1
   National Security Agency
   Research and Development Building
   R23
   9800 Savage Road
   Fort Meade, MD 20755-6000

8. CAPT Dan Galik 1
Space and Naval Warfare Systems Command
PMW 161
Building OT-1, Room 1024
4301 Pacific Highway
San Diego, CA 92110-3127

9. Commander, Naval Security Group Command 1
Naval Security Group Headquarters
9800 Savage Road
Suite 6585
Fort Meade, MD 20755-6585
ATTN: Mr. James Shearer

10. Mr. George Bieber 1
Defense Information Systems Agency
Center for Information Systems Security
5113 Leesburg Pike, Suite 400
Falls Church, VA 22041-3230

11. CDR Chris Perry 1
N643
Presidential Tower 1
2511 South Jefferson Davis Highway
Arlington, VA 22202

12. Mr. Charles Sherupski 1
Community CIO Office
Washington DC 20505

13. Ms. Deborah M. Cooper 1
Deborah M. Cooper Company
P. O. Box 17753
Arlington, VA 22216

14. Mr. Robert Wherley 1
XTS Product Technical Manager
WANG Federal Inc.
7900 Westpark Drive
McLean, VA 22102-4299

15. Mr. Paul Barbieri 1
WANG Federal Inc.
7900 Westpark Drive
McLean, VA 22102-4299

16. LCDR James P. Downey 1
DISA D6/IAESO/MSL Engineering
5600 Columbia Pike
Falls Church, VA 22041-2717

17. Susan BryerJoyner 1
22286 Capote Drive
Salinas, CA 93909

18. Scott D. Heller 1
300 Glenwood Circle #112
Monterey, CA 93940