US 20010047486A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2001/0047486 A1

Green et al. (43) Pub. Date: Nov. 29, 2001

(54) SECURE COMMERCE SERVER

(75) Inventors: **Michael W. Green**, Shoreview, MN (US); **Andrew W. Jensen**, Oakdale, MN (US)

Correspondence Address:
SCHWEGMAN, LUNDBERG, WOESSNER & KLUTH
P.O. BOX 2938
MINNEAPOLIS, MN 55402 (US)

(73) Assignee: **Secure Computing Corporation**

(21) Appl. No.: **09/850,261**

(22) Filed: **May 7, 2001**

Related U.S. Application Data

(63) Continuation of application No. 09/255,111, filed on Feb. 22, 1999, which is a continuation of application No. 08/605,320, filed on Feb. 9, 1996, now Pat. No. 5,913,024.

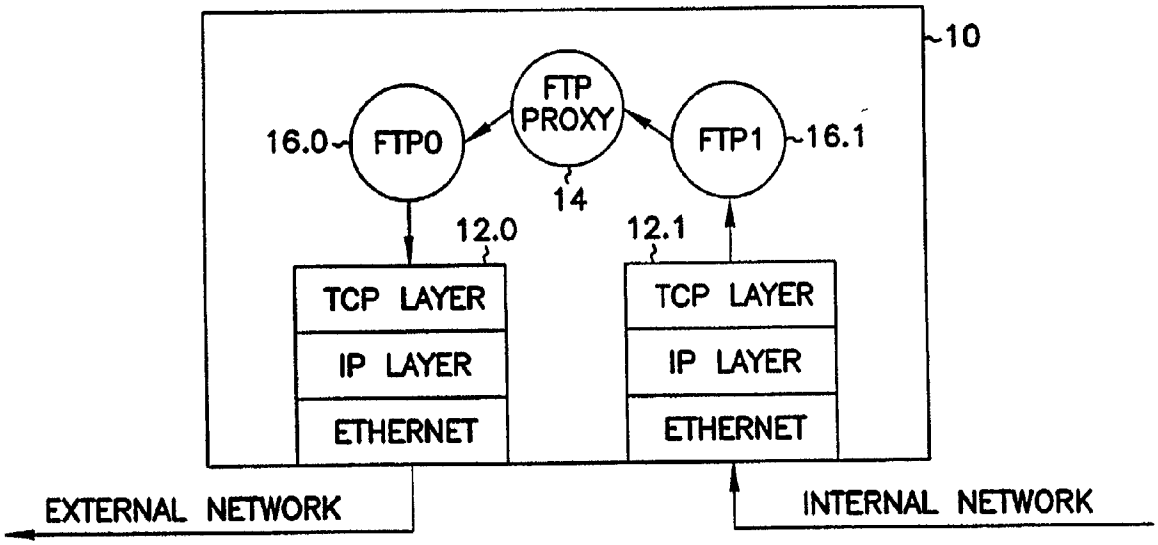
Publication Classification

(51) **Int. Cl.**[7] **H04L 12/22**; G06F 12/14; G06F 17/60

(52) **U.S. Cl.** **713/201**; 713/151; 705/44

(57) **ABSTRACT**

A secure commerce server system and method. A secure commerce server system includes a plurality of regions or burbs, including an internal burb and an external burb, a commerce server and an administration server. Processes and data objects associated with the administration server are bound to the internal burb. Processes and data objects associated with the commerce server are bound to the external burb. Processes bound to one burb cannot communicate directly to processes and data objects bound to other burbs. The administration server cannot be manipulated by a process bound to the external burb.



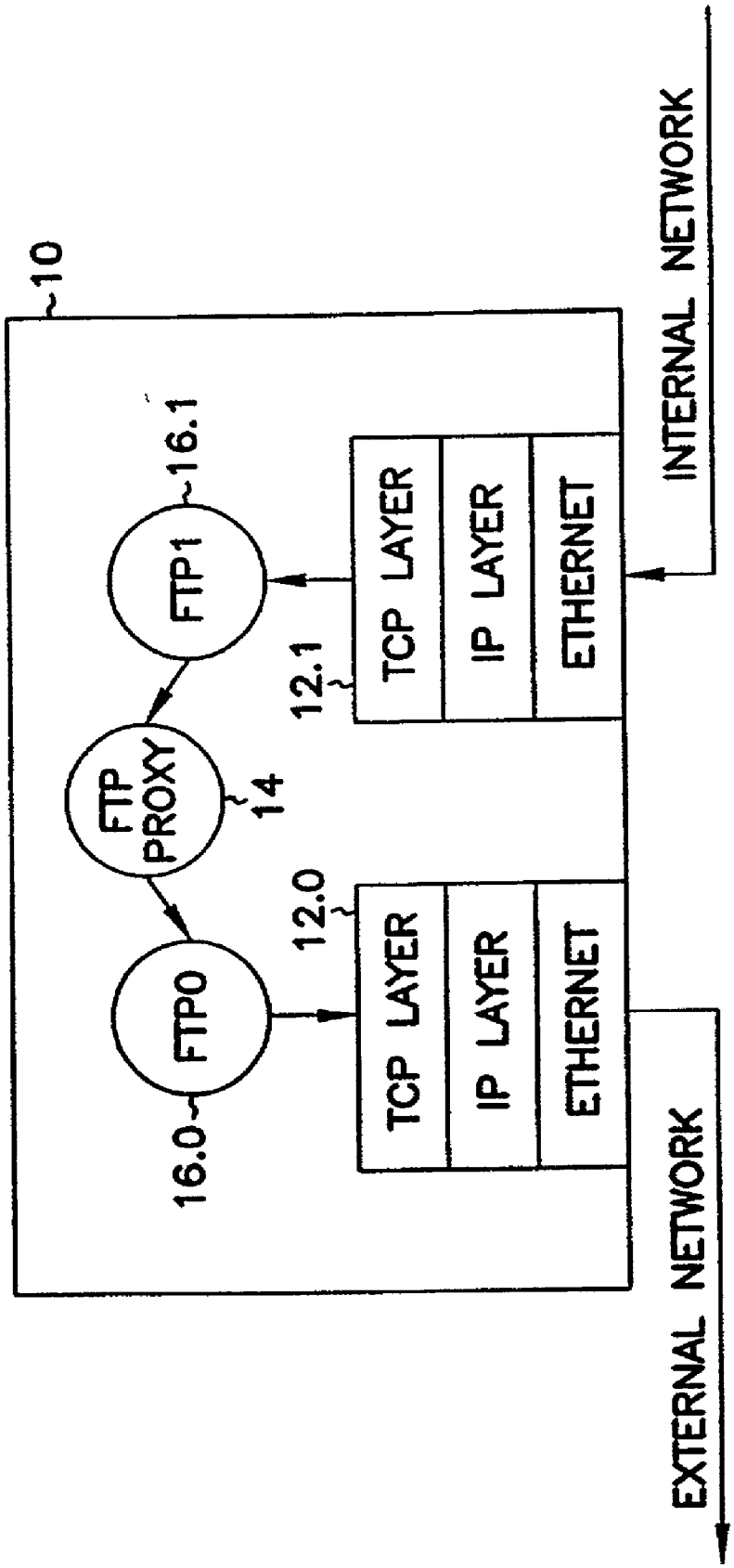

10
INTERNAL NETWORK
FTP1
16.1
FTP PROXY
14
FTP0
16.0
12.1
12.0
TCP LAYER
IP LAYER
ETHERNET
TCP LAYER
IP LAYER
ETHERNET
EXTERNAL NETWORK

FIG. 1

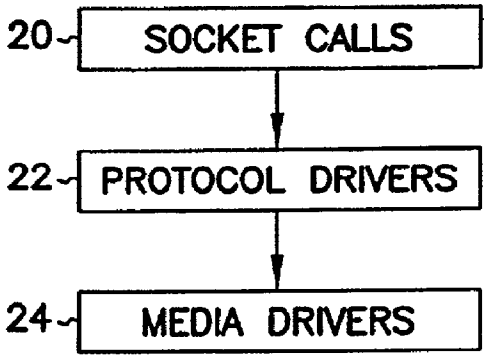

20
SOCKET CALLS
22
PROTOCOL DRIVERS
24
MEDIA DRIVERS

FIG. 2A

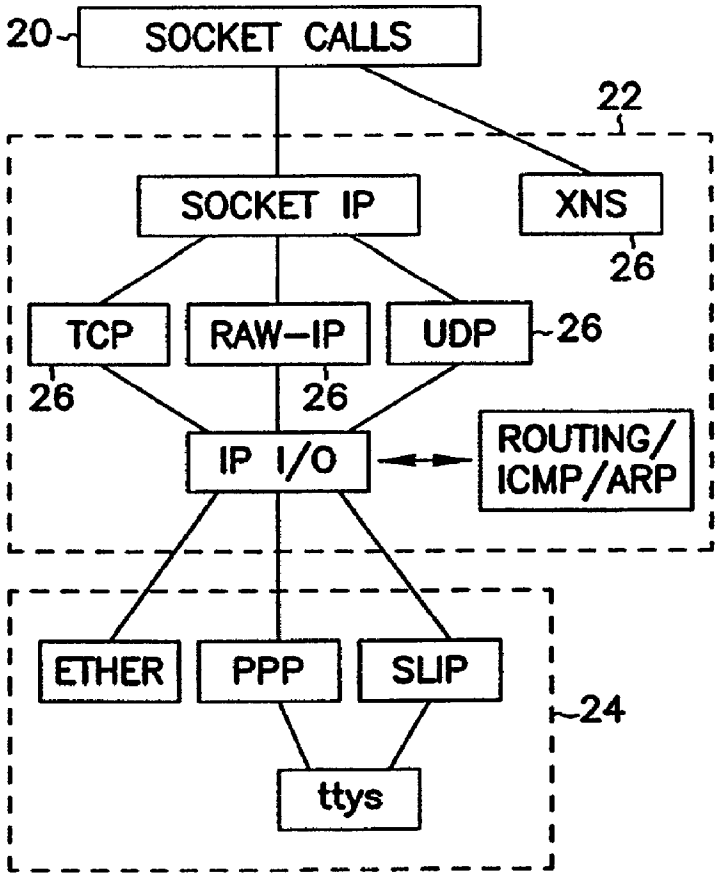

20
SOCKET CALLS
22
SOCKET IP
XNS
26
TCP
RAW-IP
UDP
26
26
26
IP I/O
ROUTING/
ICMP/ARP
ETHER
PPP
SLIP
24
ttys

FIG. 2B

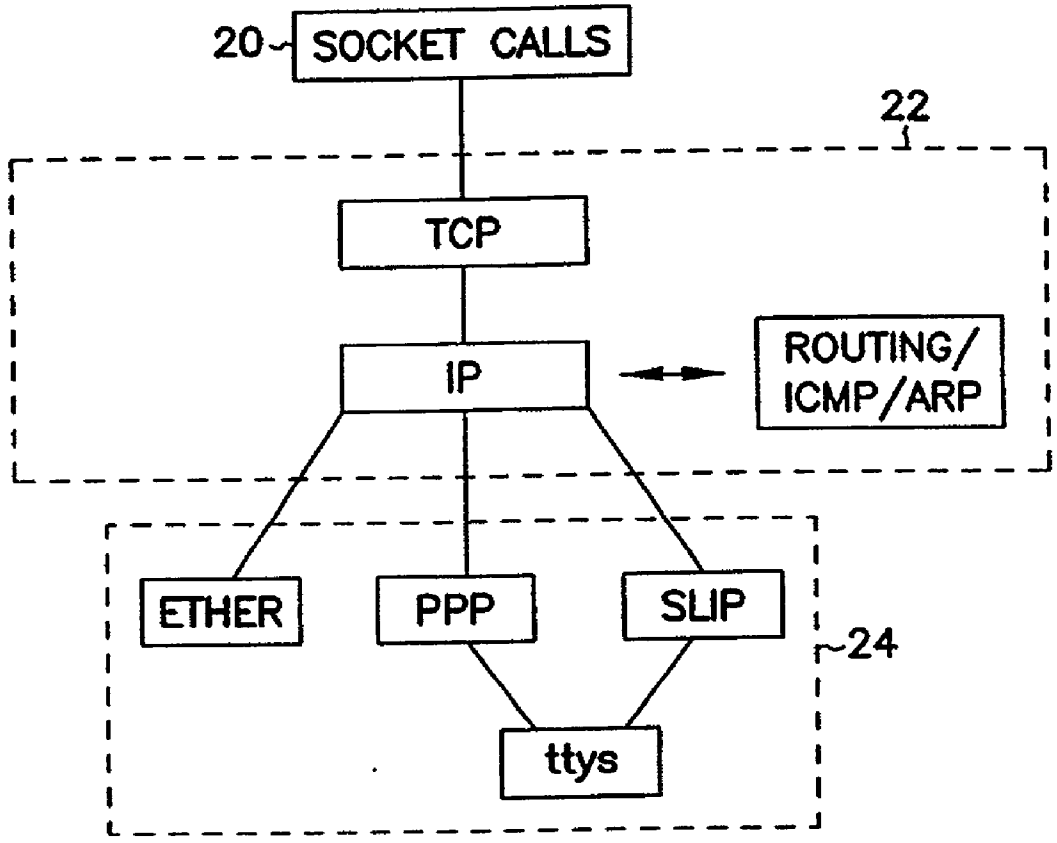

20
SOCKET CALLS
22
TCP
IP
ROUTING/
ICMP/ARP
ETHER
PPP
SLIP
24
ttys

FIG. 2C

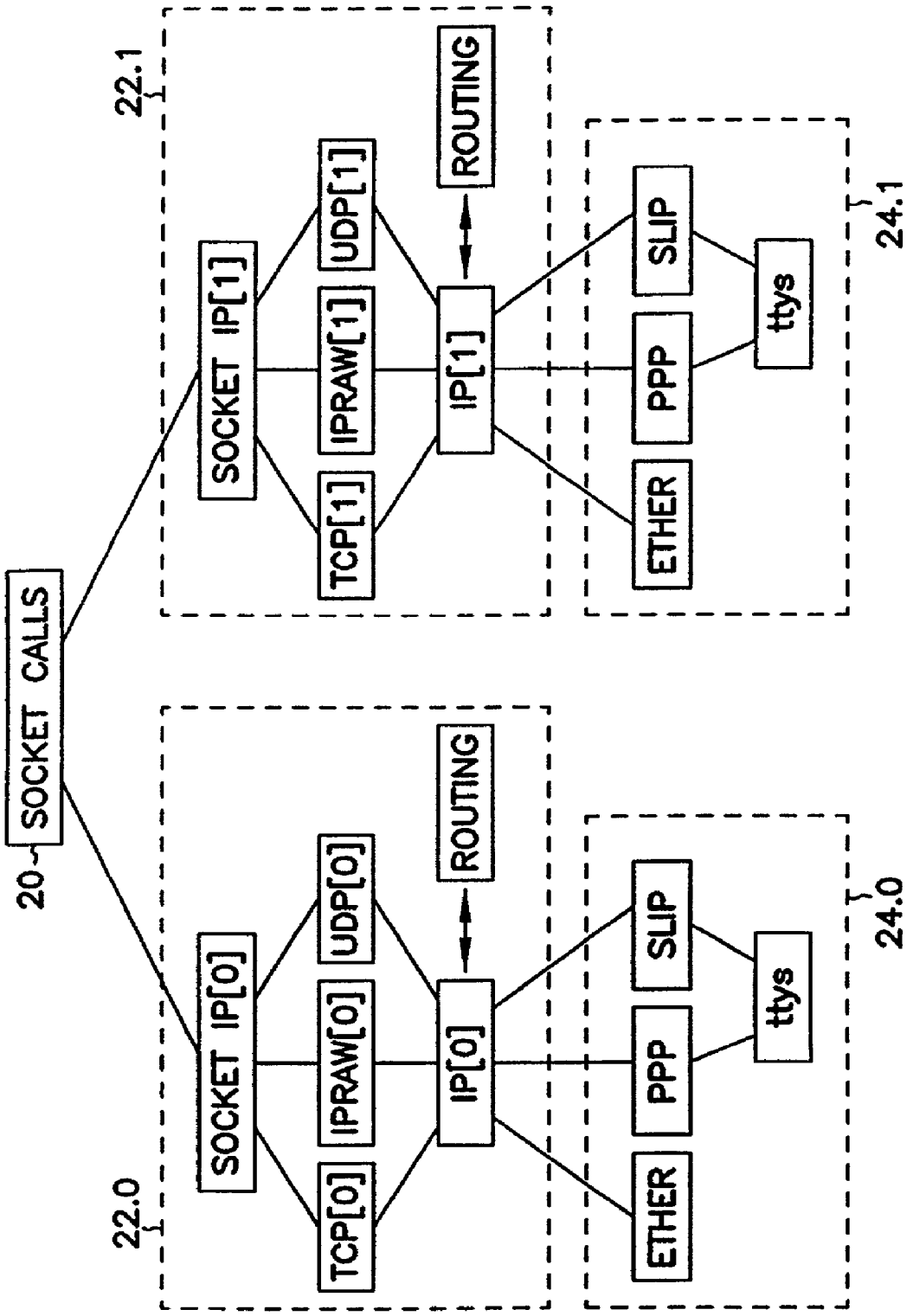

20
SOCKET CALLS
22.0
SOCKET IP[0]
TCP[0]
IPRAW[0]
UDP[0]
IP[0]
ROUTING
ETHER
PPP
SLIP
ttys
24.0
22.1
SOCKET IP[1]
TCP[1]
IPRAW[1]
UDP[1]
IP[1]
ROUTING
ETHER
PPP
SLIP
ttys
24.1

FIG. 2D

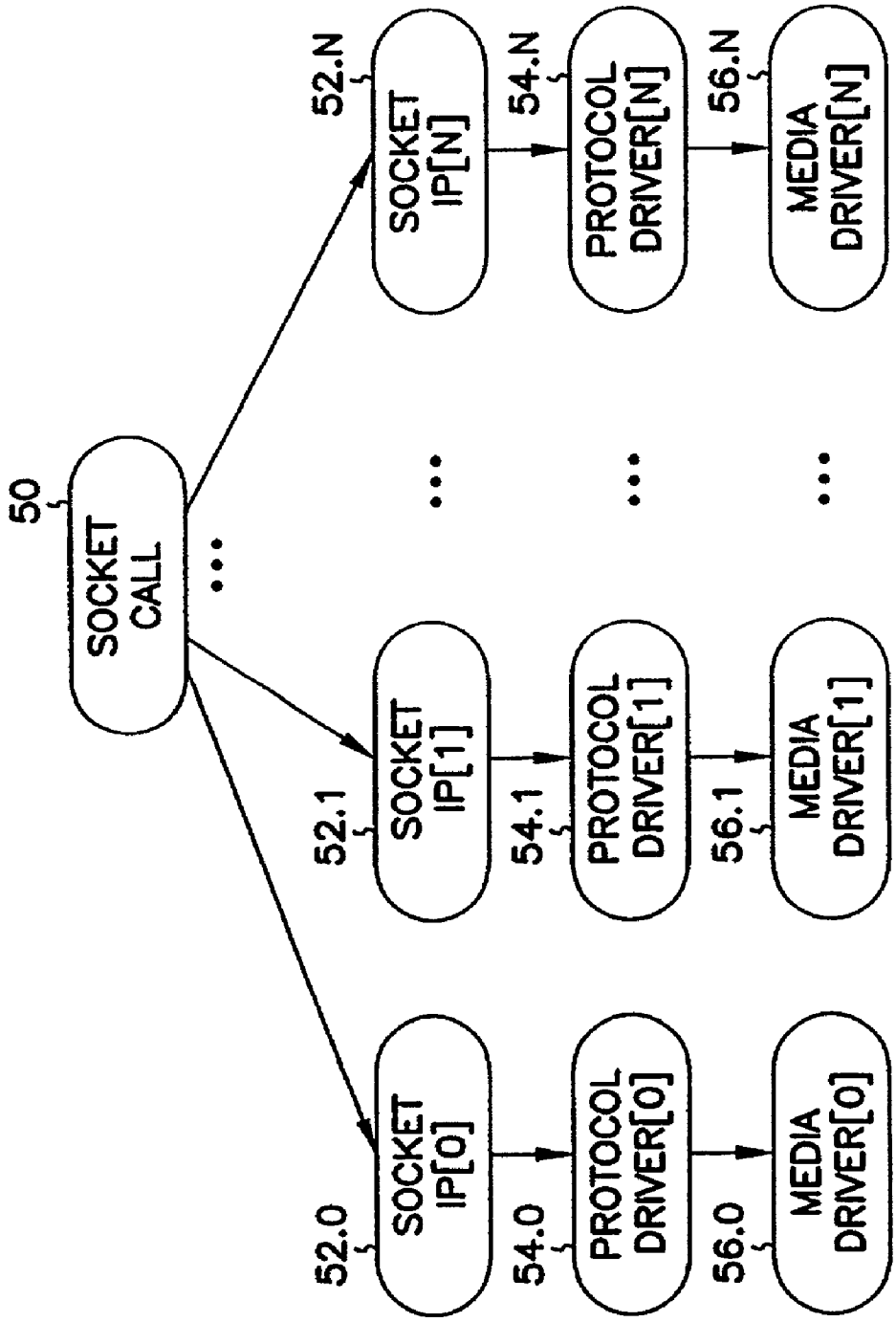

50
SOCKET CALL
52.0
SOCKET IP[0]
54.0
PROTOCOL DRIVER[0]
56.0
MEDIA DRIVER[0]
52.1
SOCKET IP[1]
54.1
PROTOCOL DRIVER[1]
56.1
MEDIA DRIVER[1]
52.N
SOCKET IP[N]
54.N
PROTOCOL DRIVER[N]
56.N
MEDIA DRIVER[N]

FIG. 3

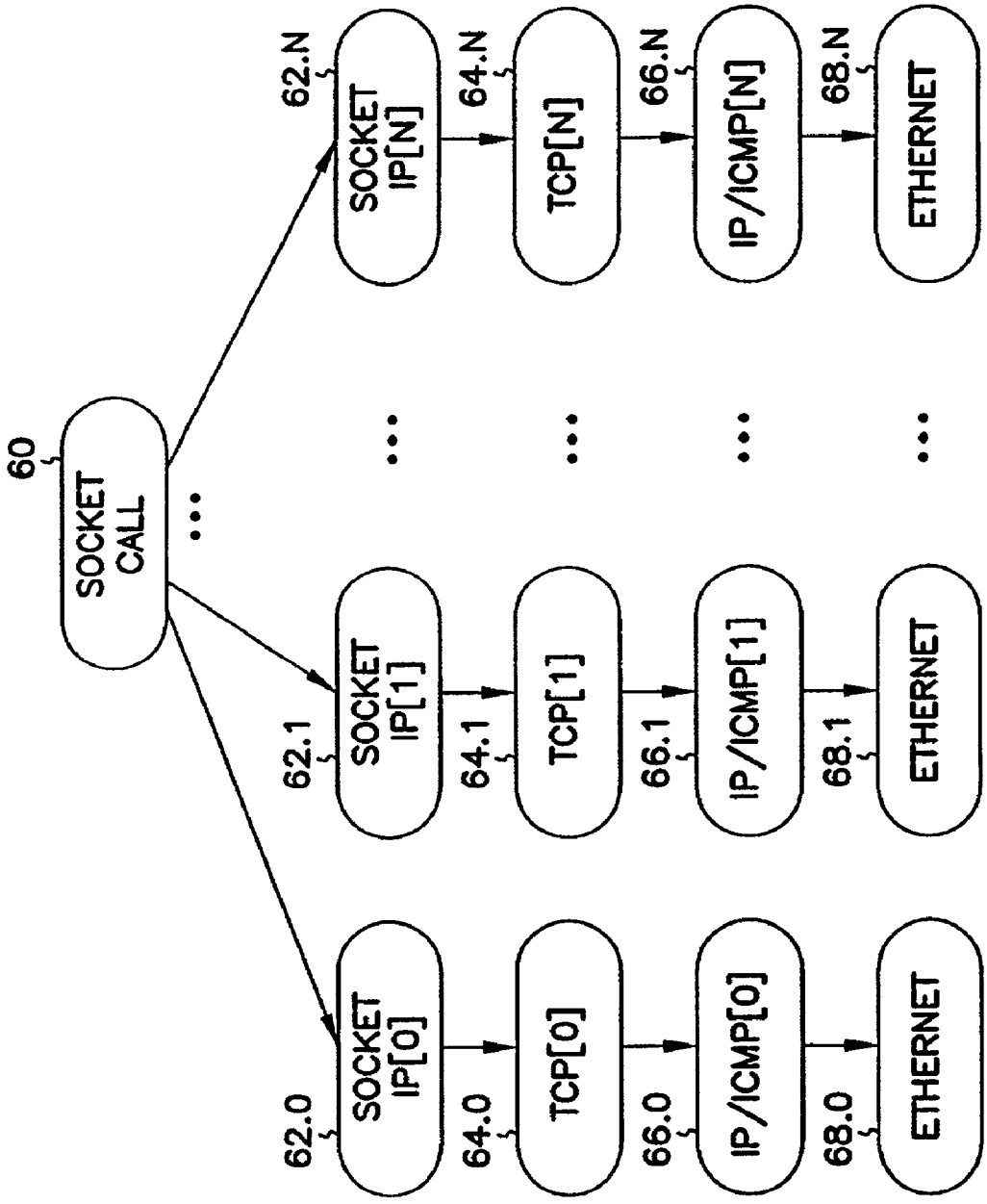

60
SOCKET CALL
62.0
SOCKET IP[0]
64.0
TCP[0]
66.0
IP/ICMP[0]
68.0
ETHERNET
62.1
SOCKET IP[1]
64.1
TCP[1]
66.1
IP/ICMP[1]
68.1
ETHERNET
62.N
SOCKET IP[N]
64.N
TCP[N]
66.N
IP/ICMP[N]
68.N
ETHERNET

FIG. 4

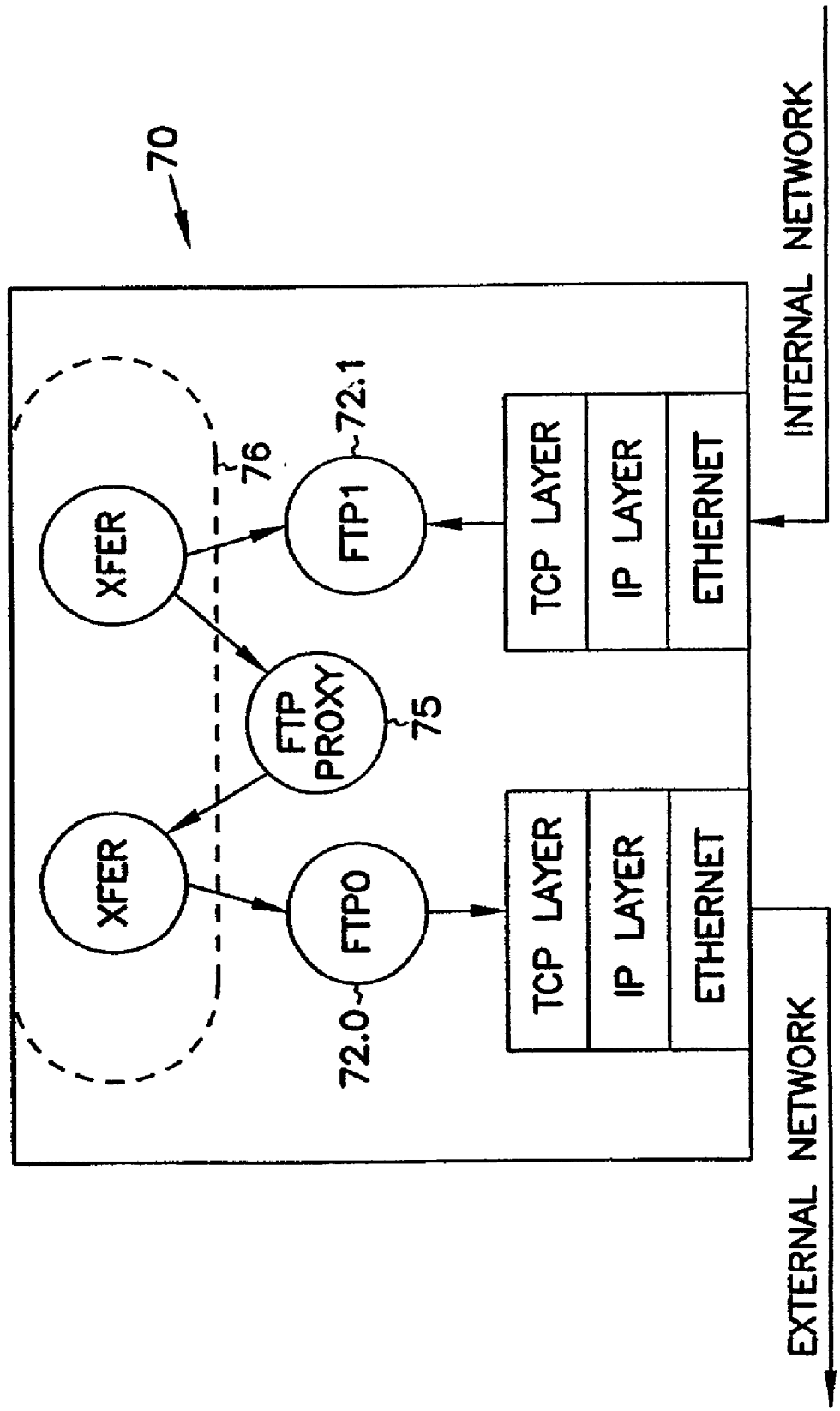
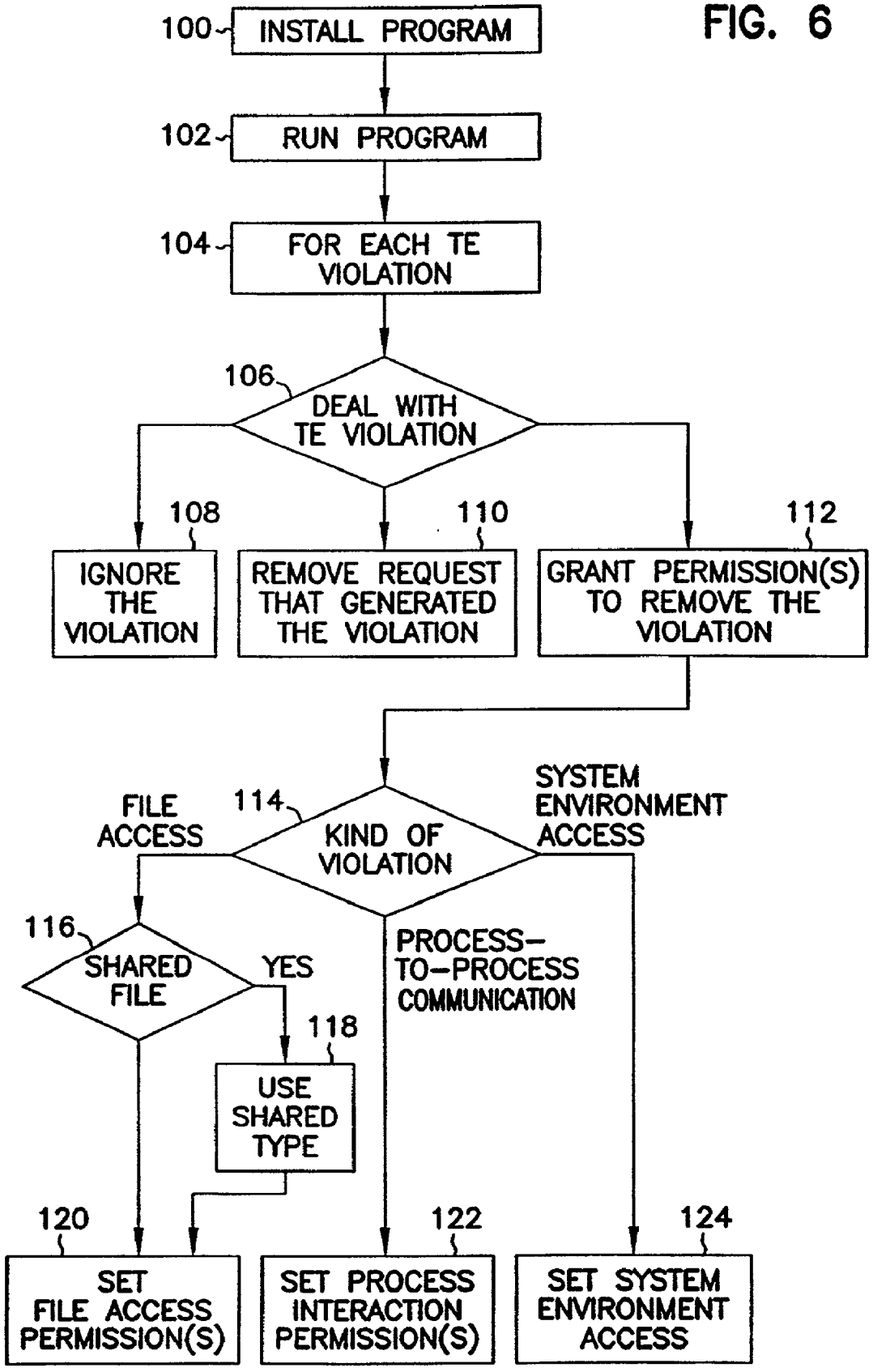

70
INTERNAL NETWORK
EXTERNAL NETWORK
XFER
XFER
FTP PROXY
75
76
FTP1
72.1
FTP0
72.0
TCP LAYER
IP LAYER
ETHERNET
TCP LAYER
IP LAYER
ETHERNET

FIG. 5

FIG. 6


100
INSTALL PROGRAM
102
RUN PROGRAM
104
FOR EACH TE VIOLATION
106
DEAL WITH TE VIOLATION
108
IGNORE THE VIOLATION
110
REMOVE REQUEST THAT GENERATED THE VIOLATION
112
GRANT PERMISSION(S) TO REMOVE THE VIOLATION
114
KIND OF VIOLATION
FILE ACCESS
SYSTEM ENVIRONMENT ACCESS
PROCESS-TO-PROCESS COMMUNICATION
116
SHARED FILE
YES
118
USE SHARED TYPE
120
SET FILE ACCESS PERMISSION(S)
122
SET PROCESS INTERACTION PERMISSION(S)
124
SET SYSTEM ENVIRONMENT ACCESS

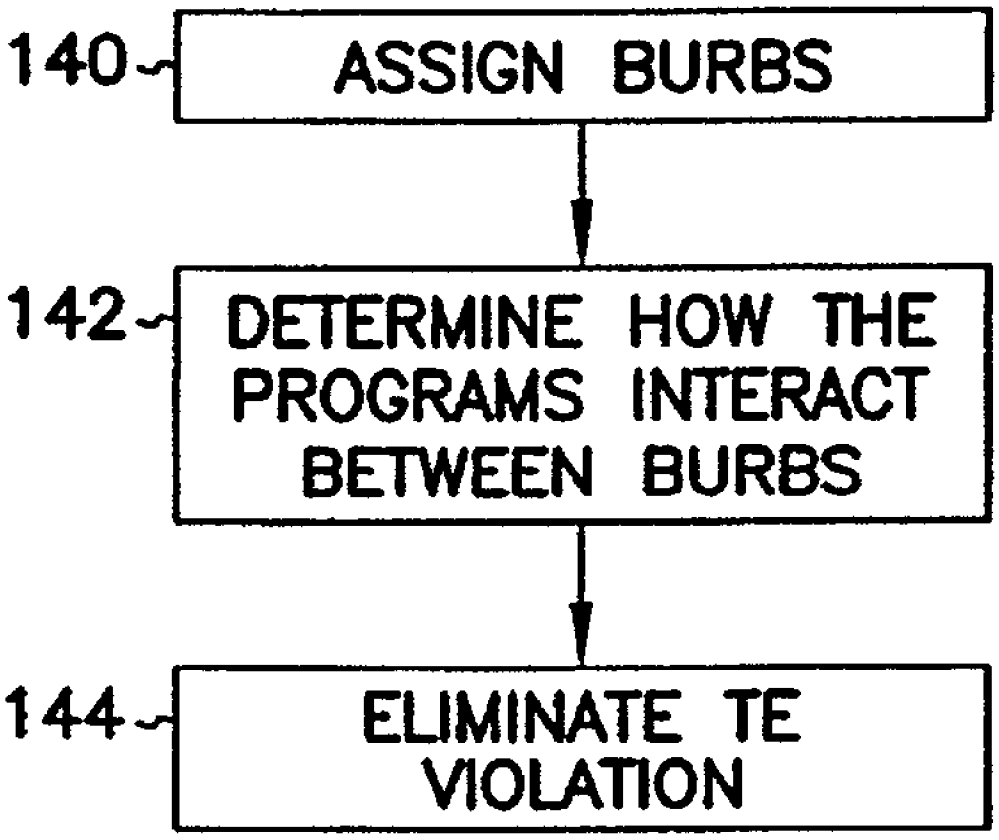

140
ASSIGN BURBS
142
DETERMINE HOW THE PROGRAMS INTERACT BETWEEN BURBS
144
ELIMINATE TE VIOLATION

FIG. 7

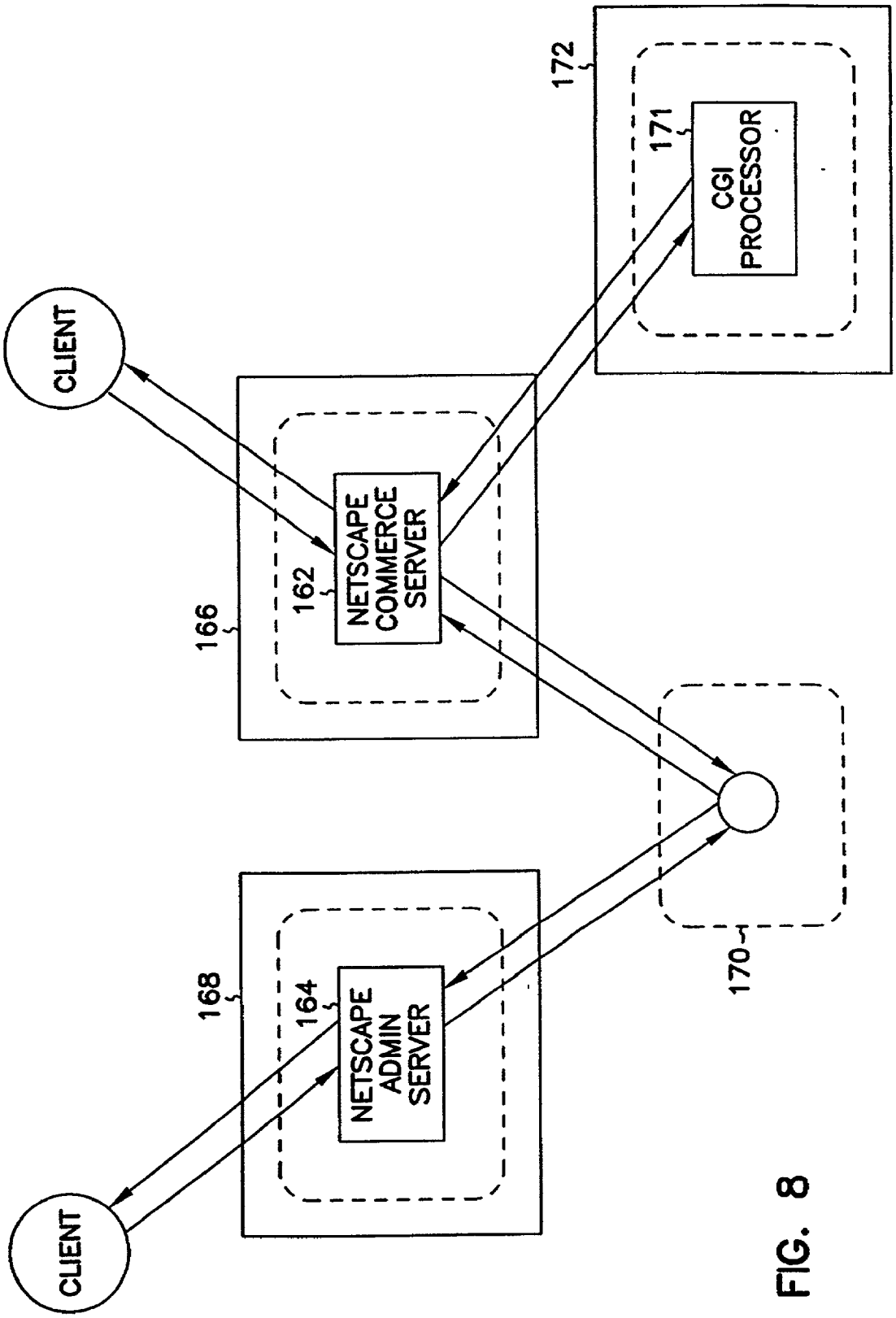

CLIENT
CLIENT
NETSCAPE COMMERCE SERVER
NETSCAPE ADMIN SERVER
CGI PROCESSOR
162
164
166
168
170
171
172

FIG. 8

SECURE COMMERCE SERVER

BACKGROUND OF THE INVENTION

[0001] This application is a continuation of U.S. patent application Ser. No. 08/605,320, filed Feb. 9, 1996.

FIELD OF THE INVENTION

[0002] The present invention relates to computer security, and more particularly, to an apparatus and method for providing increased computer security to commercial transactions across the Internet.

BACKGROUND INFORMATION

[0003] There has been an explosion in the growth of computer networks as organizations realize the benefits of networking their personal computers and workstations. Increasingly, these networks are falling prey to malicious outsiders who hack into the network, reading and sometimes destroying sensitive information. Exposure to such attacks has increased as companies connect to outside systems such as the Internet.

[0004] To protect themselves from attacks by malicious outsiders, organizations are turning to mechanisms for increasing network security. One such mechanism is described in "SYSTEM AND METHOD FOR PROVIDING SECURE INTERNETWORK SERVICES", U.S. patent application Ser. No. 08/322078 filed Oct. 12, 1994 by Boebert et al., the discussion of which is hereby incorporated by reference. Boebert teaches that modifications can be made to the kernel of the operating system in order to add type enforcement protections to the operating system kernel. This protection mechanism can be added to any other program by modifications to the program code made prior to compiling. It cannot, however, be used to add type enforcement protection to program code after that program code has been compiled.

[0005] As use of the Internet has grown, companies are increasingly interested in providing goods and services across the Internet. Software companies such as Netscape have responded by providing commerce server software. Such software typically will be partitioned into a commerce server which is accessible to the Internet shopper and an administration server which is used to maintain the commerce server and which, for security reasons, must be kept inaccessible to all but system administrators. Security mechanisms used to date have not sufficiently protected the administration server from malicious attack. What is needed is a system and method for protecting the administration servers of systems used in Internet commerce from malicious attack.

SUMMARY OF THE INVENTION

[0006] The present invention is a secure commerce server system and method. A secure commerce server system includes a plurality of regions or burbs, including an internal burb and an external burb, a commerce server and an administration server. Processes and data objects associated with the administration server are bound to the internal burb. Processes and data objects associated with the commerce server are bound to the external burb. Processes bound to one burb cannot communicate directly to processes and data objects bound to other burbs. The administration server cannot be manipulated by a process bound to the external burb.

BRIEF DESCRIPTION OF THE DRAWINGS

[0007] **FIG. 1** is a representation of a system having an internal and external interface connected via two separate protocol stacks;

[0008] **FIGS. 2*a-d*** are representations of communication protocols;

[0009] **FIG. 3** is a representation of one form of interprocessor communication which can be used in a system having a plurality of separate protocol stacks;

[0010] **FIG. 4** is a more detailed representation of one embodiment of the form of interprocessor communication shown in **FIG. 3**;

[0011] **FIG. 5** is an alternate embodiment of the system of **FIG. 1**, in which all communications between regions (burbs) pass through system space before being passed to another burb;

[0012] **FIG. 6** is a flowchart illustrating the steps taken in securing compiled program code according to the present invention;

[0013] **FIG. 7** is a flowchart illustrating the steps taken in applying network separation to compiled program code according to the present invention; and

[0014] **FIG. 8** is a representation of a system built using the steps shown in **FIGS. 6 and 7**.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0015] In the following Detailed Description of the Preferred Embodiments, reference is made to the accompanying Drawings which form a part hereof, and in which are shown by way of illustration specific embodiments in which the invention may be practiced. It is to be understood that other embodiments may be utilized and structural changes may be made without departing from the scope of the present invention. Various trademarks, including INTEL™, PENTIUM™ and UNIX™ are referred to herein. PENTIUM™ brand microprocessors are made by Intel and UNIX™ is the name of a particular type of computer operating svstem.

[0016] Computer systems which use a single communications protocol stack to handle communication between an internal and an external network are widely in use. This is the communication model used, for instance, in BSD 4.4. The problem with such a system is that once a process receives privileges within the system, it can use those privileges to access other network files. This can lead to a dangerous breach of network security. Two approaches can be used to beef up the security of such a system: type enforcement and network separation. Type enforcement adds an additional level of protection to the process of accessing files. Network separation divides a system into a set of independent regions. Through network separation a malicious attacker who gains control of one of the regions is prevented from being able to compromise processes executing in other regions. Type enforcement and network separation will be described next.

Type Enforcement

[0017] In "SYSTEM AND METHOD FOR PROVIDING SECURE INTERNETWORK SERVICES", U.S. patent application Ser. No. 08/322078 filed Oct. 12, 1994, Boebert et al. describe a way of extending type enforcement protection to a computer system having both an internal private network and an external public network. In one embodiment of such a system, a secure computer is used to connect a private network having a plurality of workstations to a public network. A protocol package (such as TCP/IP) running on the secure computer implements a communications protocol used to communicate between each workstation and the secure computer. A Local Cryptography function can be integrated into the protocol package in order to protect and authenticate traffic on the private network.

[0018] Program code running on the secure computer is used to communicate through the private network to the workstation's protocol package. In one embodiment, the secure computer is an Intel Pentium-based machine running a hardened form of BSD386 Unix. A system based on a 90 MHZ Pentium microprocessor with 32 megabytes of memory, 2 gigabytes of hard disk space, a DAT tape for backup and a CD-ROM for software loads has been found to be adequate.

[0019] Likewise, program code running on the secure computer is used to communicate through a public network interface to a public network such as the Internet. In an Internet embodiment, the program code used to communicate with the Internet is part of a set of Internet protocols which communicate with computers on the Internet through an Internet connection. In one embodiment, different protocols and cryptographic methods may be used when communicating with different entities on the Internet. In one embodiment, a top wrapper package operating in the Internet protocols is used to sit on the external, public network so that information about external probes can be logged. It is most likely that the open nature of the public network will favor the use of public-key cryptography in this module.

[0020] As noted above, in one embodiment the secure computer is an Intel Pentium-based machine running a hardened form of Berkeley's BSD386 Unix. In that embodiment, BSD386 is hardened by adding a type enforcement mechanism which restricts the access of processes to data. Type enforcement operates in conjunction with page access control bits in the virtual page translator of the Pentium to control access to objects stored in the memory of the secure computer. To accomplish this, system calls in the basic BSD386 kernel were modified so that type enforcement checks cannot be avoided. Certain other system calls were either disabled or had certain options disabled.

[0021] The type enforcement controls are enforced by the kernel and cannot be circumvented by applications. Type enforcement is used to implement data flow structures called *Assured Pipelines*. Assured pipelines are made possible by the so-called "small process" model of computation used by Unix. In this model, a computational task is divided up into small virtual units that run in parallel to each other. Unix provides a crude and loosely-controlled way of sharing data between processes. Type enforcement supplants this with the rigorously controlled, configurable structure of assured pipelines. It should be noted that Type Enforcement works best as a supplement to the normal Unix permissions. That is, the Unix permissions are the first line of defense; when processes get past the Unix permissions, however, they run into the type enforcement checks.

[0022] In one embodiment, the secure computer has been configured under BSD386 to run in one of two states: administrative and operational. In the administrative state all network connections are disabled and the Server will only accept commands from a properly authenticated System Administrator accessing the system from the hard-wired administrative terminal. This feature prevents anyone other than the System Administrator from altering the security databases in the secure computer.

[0023] In the operational state the network connections are enabled and the Server will execute only software which has been compiled and installed as executable by an assured party.

[0024] The two states are reflected in two separate kernels. The administrative kernel is not subject to type enforcement. Instead, it is network isolated and accessible only to authorized personnel. This means that in administrative kernel mode, the secure computer cannot be seeded with malicious software by any but the people charged with system administration.

[0025] On the other hand, the operational kernel is subject to type enforcement. This means, for instance, that executable files stored in the memory of the secure computer cannot be executed without explicit execution privileges. In one such embodiment, executable files cannot be give execution privileges from within the operational kernel. Instead, the secure computer must enter administrative kernel to grant execution privileges. This prevents execution of malicious software posted to memory of the secure computer. Instead, only executables approved by operational administrators while in administrative kernel mode ever become executable within operational kernel mode of the secure computer. In one such embodiment, administrative kernel can be entered only from either a manual interrupt of the boot process to boot the administrative kernel or by booting the secure computer from a floppy that has a pointer to the administrative kernel.

[0026] The flow of data between processes is limited to transfers through assured pipelines controlled, in one embodiment, by the access enforcement mechanism of the Intel Pentium processor. Virtual memory translation circuitry within the Pentium processor includes a mechanism for assigning access privileges to pages of virtual memory. This ensures that control is imposed on every fetch from, or store to, the machine memory. In this way, the protection is made continuous. The Pentium access control mechanism enforces the following modes of access:

[0027] Read Only (R): Data values may be fetched from memory and used as inputs to operations, but may not be modified or used as program text.

[0028] Read Execute (RE): Data values may be fetched from memory and used as inputs to operations, and may also be used as program text, but may not be modified.

[0029] Read Write (RW): Data values can be fetched from memory and used as inputs to operations, and may also be stored back in modified form.

[0030] No Access: The data cannot be fetched from memory for any purpose, and it may not be modified.

[0031] These hardware-enforced accesses can be used to force data flowing from the internal private network to the Internet to go through a filter process, without any possibility that the filter is bypassed or that filtered data is tampered with by possibly vulnerable software on the Internet side of the filter.

[0032] The access a process has to a data object via type enforcement is defined by an entry in a central, protected data structure called the *Domain Definition Table* (DDT). A Domain name denotes an equivalence class of processes. Every process in execution has associated with it two Domain names which are used to control its interaction with objects and with other Domains. The real Domain of a process is used to control Domain to Domain interactions and to grant or deny special, object-independent privileges. The effective Domain of a process is used to control its access to objects. The real and effective Domains of a process will generally be identical; the circumstances in which they differ are described below.

[0033] A Type name denotes an equivalence class of objects. Objects are, in general, the "base types" of BSD/386 Unix: files, directories, etc. There are eight default subtypes-:file, directory, socket,jfifo, device,port, executable, and gate. The implied default subtype pipe is, in effect, untyped because no check is made on access to pipes. Type names consist of two parts and, in the preferred embodiment, are written in documentation and comments as creator:subtype. The creator field is the four-character name of the Domain which created the object. The subtype field denotes the "class" of the object within that Domain. Subtype names are also four characters long and may contain any printable character except |*| or whitespace.

[0034] Subtypes will not be shared; thus Mail: file means, in effect, "the files private to the Mail Domain." When objects are created they are automatically assigned the appropriate default subtype. Objects which are to be shared between Domains must have their subtype changed from the default to an explicit subtype.

[0035] Subtypes can be assigned one of three ways:

[0036] By having a default subtype assigned when the object is created by the operational kernel.

[0037] By having an explicit subtype assigned by the privileged chtype or fchtype syscalls. Thus a file which was to be shared between the Mail Domain and some other Domain would first be created as Mail:file and then changed to, e.g., Type Mail:Publ. If a subtype is changed to a default subtype, then the object becomes private.

[0038] By having a default or explicit subtype assigned administratively by the administrative kernel.

[0039] The default subtypes exec and gate are "static." The operational kernel will not create any objects of those subtypes, change those subtypes into any other subtype, or change any other subtypes into a gate or exec.

[0040] The Domain/Type relationship is used to define the modes and consequences of accesses by processes to objects. The modes and consequences of accesses are defined by access attributes which are store in the DDT database. The DDT database is "indexed" by three values:

[0041] The effective Domain of the process requesting the access or action.

[0042] The creator field of the object Type.

[0043] The subtype field of the object Type.

[0044] The result of "indexing" is the retrieval of a set of access attributes. The term "attribute" is used instead of "mode" because some of the attributes define immediate side effects. The selection of attributes was governed by the following considerations.

[0045] To constrain the modes of access which processes may exercise on objects.

[0046] To prevent the execution of any application software other than that which has been installed through the controlled administrative environment.

[0047] To enable the spoofing of attackers so that the attack response facilities can be used to trace them at the physical packet level. This required a more sophisticated response to illegal accesses than just shutting down the offending process.

[0048] Gating permits a process to temporarily become a member of another Domain. The "home" or permanent Domain of the process is called its real Domain and the temporary or assumed Domain is called the effective Domain. Implicit gating is used when it is necessary to strictly control the manner in which the effective Domain's accesses are used. Implicit gating "ties" the temporary Domain change to a specific executable which has been subjected to extra scrutiny to insure that the effective Domain's accesses are used safely. The "tying" of the Domain change is done because the Domain change is a side effect of execve'ing a special executable: one whose subtype is gate. Implicit gating also allows Domain changes to be defined by changing the Type of an executable instead of inserting explicit calls into the source code.

[0049] Explicit gating is used when a looser control on the temporary Domain transition is appropriate, or when the "tying" of the gating to a specific executable would require excessive restructuring of existing software.

[0050] Domain changes are controlled by the DIT. The logical structure of the DIT is a table with an entry for each Domain. The logical structure of each entry is that of two pointers, one to a list of allowed real Domains and the other to a list of allowed effective Domains. Thus, if a process executed a makedomain or changedomain, the real Domain of the process selects the entry and the Domain given by the domainname argument must be on the list of allowed real Domains for the Domain change to happen. Likewise, if a process executes a gate, the Domain given in the domainname argument must be on the list of allowed effective Domains. Finally, if a process executes an execve of an executable whose subtype is gate, the creator Domain of that executable must appear on the list of allowed effective Domains.

[0051] Certain kernel syscalls are restricted to processes executing out of privileged Domains. In one embodiment two levels of checks are made. First, the normal BSD UNIX permissions are checked; if these permissions cause the

operation to fail, the system call returns the normal error code. If the UNIX permissions are adequate, the type enforcement (TE) privileges are checked next, (and thus in addition to the UNIX permissions).

[0052] The following BSD system calls have been modified to properly implement type enforcement. The modified calls have been grouped into four groups for ease of explanation.

[0053] The first group of system calls that require modification are those that set or affect the identity and/or state of the computer. Two of these system calls affect the computer's internal time: settimeofday and adjtime. Both of these system calls have been modified to require the <can_set_clock> privilege before the request will be honored. In the event of a privilege violation, the system call will raise an Alarm, will not honor the request, but will return success.

[0054] Other system calls which affect the computer's notion of self identity are sethostname and sethostid. Both of these system calls have been modified to require the <is-startup> privilege before the request will be honored. In the event of a privilege violation, the system call will raise an Alarm, will not honor the request, and will return the EPERM error flag. The last system call affects the computer's runtime status, reboot. The reboot system call has been modified to require the <admin-reboot> privilege before the request will be honored. If the request is honored, the computer will boot to the admin kernel (single-user mode only with networking disabled). In the event of a privilege violation, the system call will raise an Alarm, will not honor the request, and will return the EPERM error flag.

[0055] The second group of system calls that require modification are those that allow interaction with the computer's file system. The open system call has been modified to become the primary TE check. After performing the normal BSD UNIX permission checks, the TE check is performed. An Alarm is raised if the TE check returns null (no permissions), or if the caller asks for read but the <ddt_read> privilege is not set, or if the caller asks for write but the <ddt_write> privilege is not set. The creat system call has been modified to set the new file's Type to <creator:file>. Additionally, the creation of a new file implies a write operation on the directory, which in turn implies that the TE-modified open system call will be used to open the directory file, which in turn implies that TE can be used to control the success or failure of the creat system call. The unlink and rename system calls are modified in like manner. The unlink system call requires the <ddt_ destroy> privilege. The rename system call requires the <ddt_rename> privilege on the "from" file, and if the "to" file exists, it further requires the <ddt_destroy> privilege on the "to" file. In the event of a privilege violation, both the unlink and rename system calls will raise an Alarm, will not honor the request, but will return success. The access system call is modified to require the <mode> privilege on the file pointed to by the path. In the event of a privilege violation, the access system call will raise an Alarm, will not honor the request, but will return success. The chflags, fchflags and quotacl system calls are modified in like manner. All are modified to perform no functions. Attempts to call them will raise an Alarm, will not honor the request, and will return EPERM. The mknod system call is modified to perform no function. Attempts to call it will raise an Alarm, will not honor the request, and will return EPERM.

[0056] The third group of system calls that require modification are those concerning process creation, maintenance and tracing. The fork system call has been modified so that the child process inherits both the real and effective Domains of the parent process. The execve system call is modified to require the <ddt_exec> privilege on the file pointed to by the path before the request will be honored. The real and effective Domains of the process remain unchanged. In the event of a privilege violation, the system call will raise an Alarm, will not honor the request, but will return success. The ktrace, pirace and profil system calls are modified in like manner. All are modified to perform no function. Attempts to call them will raise an Alarm, will not honor the request. The ktrace and ptrace system calls will return EPERM, whereas the profil system call will return EFAULT.

[0057] The mprotect system call is modified to perform no function. Attempts to call it will raise an Alarm, will not honor the request, and will return EPERM.

[0058] The fourth group of system calls that require modification are those that relate processes to user ids. The setuid and seteuid and old.seteuid system calls are modified in like manner. All are modified to require the <suppress_su alarm> privilege before the request will be honored. In the event of a privilege violation, the system call will raise an Alarm, will not honor the request, and will return success. The acct system call is modified to perform no function. Attempts to call it will raise an Alarm, will not honor the request, and will return EPERM. The setlogin system call is modified to require the <can_setlogin> privilege. In the event of a privilege violation, the access system call will raise an Alarm, will not honor the request, but will return success.

[0059] A final set of system calls consists of those that are removed entirely from the BSD UNIX kernel. This set of system calls includes: obs vtrace, nfssvc, asynch_daemon, getfh, shmsys, sfork, getdescriptor, and setdescriptor.

Network Separation

[0060] The goal of network separation is to provide an operating system kernel with support for multiple networking protocol stacks. A system **10** having such an operating system kernel is illustrated in **FIG. 1**. System **10** is split into separate regions, with a domain **16** and a protocol stack **12** assigned to each region. Each protocol stack (**12.0**, **12.1**) has a fixed set of interfaces bound to it. For example, two or more Ethernet drivers can be connected to, for instance, protocol stack **12.0**.

[0061] A given socket will be bound to a single protocol stack **12** at creation time. Each protocol stack **12** will have its own independent set of data structures including routing information and protocol information. No data will pass between protocol stacks without going through proxy space and being sent back down another protocol stack by a proxy program **14**. Proxy **14** acts as a go-between, therefore, for transfers between domains **16.0** and **16.1**. No user applications will have direct access to either network.

[0062] The embodiments discussed will only cover common networking support, such as the media layer drivers like PPP, Ethernet, and SLIP and the TCP/IP suite of protocols. The BSD kernel supports other protocols such as CCITT/X.25 and XNS. These are not covered here, although it should be apparent that network separation can be extended

to other protocols by dividing the protocol stacks associated with the protocol layers into multiple protocol stacks with the number of (and the name of) stacks being the same across all protocol suites.

[0063] The following terminology will be used throughout the document: BSD The BSD/OS 2.0 Unix operating system as based upon the BSD 4.4 Lite distribution. In the case of the networking functionality the code is virtually identical across all 4.4 derived Unix systems (NetBSD1.0, FreeBSD 2.0, and lBSD/OS 2.0). The only significant differences are the hardware device drivers in terms of how they autoconfig during boot, which ones are present, and their internal code. Their interface to the rest of the networking code is effectively identical. BSD kernel The BSD/OS 2.0 kernel.

[0064] Burb The aggregation of a protocol stack with all the processes that can access that stack. Processes that can access a particular protocol stack are said to be bound to that protocol stack.

[0065] NBURBS The number of protocol stacks or network interfaces to which all networking processes and related entities are bound.

[0066] Kernel space The address and code space of the running kernel. This includes kernel data structures and functions internal to the kernel. Technically the kernel can access the memory of the current process as well, but "kernel space" generally means the memory (including code and data) that is private to the kernel and not accessible by any user process.

[0067] User space The address and code space of processes running that isn't kernel space. A running process will often execute inside of the kernel during system calls, but that is kernel space since the kernel *always* is running within the context of the current process except during the few moments of the actual context switch—in a kernel without SMP support and kernel threads. In general, user space refers to code and data allocated by a code that is loaded from an executable and is available to that process, not counting the kernel private code data structures.

[0068] Protocol stack The set of data structures and logical entities associated with the networking interfaces. This includes sockets, protocol drivers, and the media device drivers.

[0069] Link level & hardware drivers These are the (almost in some cases) bottom layer drivers that talk a particular physical and/or link level protocol (Ethernet, PPP, SLIP). They may be a hardware driver, in the case of most Ethernet drivers, or they may sit on top of yet other drivers, such as PPP on a TTY on the COM port driver or even a parallel port Ethernet driver on top of the LPT port driver. Most of these drivers have two layers, a generalized layer that handles the common parts of the link level protocol (Ethernet, PPP, etc.) and the hardware specific driver.

[0070] One of the goals of a secure operating system kernel is to allow dividing the network interfaces into distinct regions so that there is assurance that packets are never quietly passed through the kernel between those regions. In one embodiment, the following rules must be met to ensure secure control of packet transfer between regions:

[0071] A single user process can only send and receive information (packets) from one region at a time.

[0072] Once a process is bound to a region, it can only access that region.

[0073] Incoming packets can only go to processes that are in the region associated with the interface the packet arrived on.

[0074] Any data passing through the computing system to a different region must come into user process and be handed off to a different process that has access to the other region, a proxy program, to be sent out again.

[0075] The reasons for these design requirements are fairly simple. The goal is that any information passing through the computer between different regions has to be passed through a predefined, or assured, pipeline.

[0076] There are three ways of achieving this goal:

[0077] 1) Packet Filtering Packet filtering would be done using conventional approaches similar to current firewalls and screening routers. This could be enhanced with additional filters based on interface rather than address to prevent certain types of spoofing.

[0078] 2) Packet Separation A given message, incoming or outgoing, would be assigned a Type based on the interface it arrived on or the socket it was sent from. The packet would then be thrown out at the top or bottom level if that Type didn't match the Type of the interface it was being sent on.

[0079] 3) Separate Protocol Stacks The protocol stacks would be separated into multiple instances with a given interface or socket existing on only one.

[0080] In arriving at our current approach, the packet filtering solution was quickly discarded. Although you could eliminate several kinds of spoofing, many of them would still remain. For filtering to work at all, IP packet forwarding must be enabled, but then you are vulnerable to all sorts of nasty things sneaking through the computer system using the normal sorts of attacks with various forms of spoofing, piggybacking, and just plain misconfiguration of complex filtering expressions.

[0081] The packet typing approach had merit, but required extensive code changes as all of the various layers would have to be rewritten to handle additional fields in all of the various structures (mbufs, sockets, etc.). The other problem with the packet typing approach was how to deal with kernel internally generated messages like ICMP messages.

[0082] The separate protocol stack approach was selected because it gives a very clear split of the regions making assurance easier. It also requires surprisingly few code changes. Rather than change the existing data structures, they are simply replicated as needed for the various regions. The initial reference has to be fixed, but most references are done by cached pointers within the various structures, so many of the functions and layers of the protocol stack require no changes at all. For instance, the hardware drivers require absolutely no changes. The socket code requires only a minor change during the socket instantiation; subsequent references are done using the already fixed pointer to the particular protocol driver. This also gives a performance win

because basically no lookups are done dynamically, only an extra level of indirection during initialization, but not subsequently.

[0083] The general structure of the design chosen involves duplicating all of the protocol stacks where each stack is independent of the others. The routing to a given stack is done at the very top or bottom of the dataflow so that a given packet, piece of data, control message, etc. is bound to a particular stack at creation.

[0084] A name was needed for each of these protocol stacks, and for the processes and related entities bound to them, the name that was chosen was burb.

[0085] Second, the decision whether to replicate and subdivide into separate burbs is made on a protocol family by protocol family basis. Generally all external supported networking protocol families will be divided. The internal Unix domain socket family won't be replicated. This is important because the Unix Domain sockets (AF_UNIX) will continue to exist in one common region for the entire system. This feature becomes important for the pipeline structure given later.

[0086] To identify the network/socket access a process is given a burb ID that says which burb it is in. In one embodiment, the burb ID is an ordinal number.

[0087] The general picture of the protocol stacks in a normal BSD kernel is shown in **FIG. 2***a*. An expanded view of this picture to include the various IP protocol drivers is shown in **FIG. 2***b*.

[0088] When a socket is actually created, it is explicitly, during the initial socket() call **20**, bound to one of the protocol drivers **22** via a pair of pointers in the socket structure, the soproto member (a pointer to a struct protosw), and the so_pcb member. The protosw structure is unique per protocol driver, but there is only one per protocol driver. The sorpcb pointer is an opaque pointer to a socket specific structure that has the protocol specific information for a socket (such as the TCP state information for a TCP socket).

[0089] At the bottom level, the generic media drivers **24** (Ethernet, PPP, SLIP) put incoming packets on a per protocol family (IP, XNS, CCITT, etc.) queue, ipintrq in the case of IP.

[0090] The picture then for a created socket becomes something like the system of **FIG. 2***c*.

[0091] So, the fundamental concept of the kernel portion of the design is that the protocol stacks are replicated N times (with N being a fixed small constant, typically 2-10), with all protocol level and other shared data structures being replicated. In practice, each burb/protocol stack has a name that maps onto a number, starting at 0, that is used as an index. Each static structure is converted into an array. This also makes it easy to change the number of burbs supported. **FIG. 2***d* illustrates a system having two burbs and limited to just IP. A generic system having N burbs is shown in **FIG. 3**. A specific TCP/IP system having N burbs is shown in **FIG. 4**.

[0092] Once the socket is created then the picture becomes much simpler again, because the individual data structures are then bound explicitly and no lookup or search stage occurs again. All activity then occurs within one of the burbs (e.g., TCP[0]).

[0093] The advantage of duplicating the protocol stacks in their entirety is that the number of data structures to be duplicated, and the points that those are referenced, is manageable, much more so that making a given datum have a type and/or domain and having to "route" within the kernel based on that at every layer. As mentioned before, the basic approach is that all shared data structures are replicated N times, 1 for each burb. These data structures are a reasonably finite list. They include: Protocol Domain List and Tables (domains, protosw)—When a socket is created, a family and a type is specified, such as socket (AF_INET, SOCK_STREAM). The socket creation routine first looks up the top level protocol in the family list, ala: AF_INET, which returns a second level table of protosw structures that has the S individual protocol drivers. The second level table is in the form of XXX domain, such as inetdomain. The first level list is called "domains".

[0094] The design replicates only the tables (when that family supports separate burbs, such as IP, route). In the case of things like the AF_UNIX domain, each list will point at the same table. The lookup will then return a protosw structure unique to the instance of the protocol driver, such as inetsw[proto][burb], instead of inetsw[proto]. The routing structure will be replicated into routesw [NBURBS].

[0095] Protocol Interrupt Queues (ipintrq)—External networking protocols each have an input queue for incoming packets. Each queue is a simple mbuf list. So queues for protocols will be replicated, becoming things like ipintrq [NBURBS] instead of just ipintrq.

[0096] IP Fragmentation Queues (ipq)—Similar to the IP interrupt queue, there is a reassembly queue, ipq, to reassemble fragmented IP packets. All network interfaces have a maximum transmission unit (MTU). Outgoing packets larger than the MTU are fragmented by the IP layer into smaller packets for transmission. When the fragmented packets arrive at the destination, they are placed on a reassembly queue waiting for reassembly. This queue is also replicated into ipq [NBURBS].

[0097] Interfaces (ifnet)—Each interface, including the loopback will be bound at boot time to a given burb by a one time system call. A burb field will be added to the ifnet structure, ifnet.if_burb, which is unique per interface so that it knows which burb it's in and which interrupt queues to send data to. The binding will also be one shot, so that once an interface is bound, it can't be rebound to a different burb.

[0098] Other data structures are already unique per interface. The other item to be replicated is the list of interfaces. There will be one list per interface. The interface binding call will set the burb of the interface and tell it which interface list to put itself on during initialization.

[0099] Loopback Interface (loif)—The loopback interface IP address will have to be unique per interface. In other words, the recommended loopback address 127.0.0.1 cannot be shared between the interfaces. The loopback interface data structure will be replicated, loif [NBURBS]. Loopback addresses will be subnetted, with netmask 255.255.0.0, to give each interface a unique IP address, 127.burb.0.1.

[0100] Protocol Control Blocks (tcb, udb, raweb)—Each protocol maintains its own list of protocol control blocks, which hold various pieces of information required for the

socket operations. These lists will be replicated and maintained on a burb basis: tcb [NBURBS] for TCP, udb [NBURBS] for UDP, and rawcb [NBURBS] for routing sockets.

[0101] Routing Tables (rt_tables)—There is a single master routing table, rt_tables [family], that is a linked list of routes. This table will be separated into one table per burb, rt_tables [family][NBURBS]. There will be no master routing table.

[0102] These routing tables will be used as lookup tables so that the incoming end of a proxy knows which outgoing proxy to connect to. For outgoing packets, the search always starts with the local table. If a route cannot be found, then the system searches other tables. If a route is found on a non-local table, then the networking code will hand the packets off to the proxy processes.

[0103] ARP Tables and Queues (llinfo_arp, arpintrq)—There are no data structures at the common link level (Ethernet, PPP, SLIP) except the ARP table and the ARP interrupt queue, llinfo_arp and arpintrq. This again is a list of interfaces with just the information needed for ARP and a queue for incoming ARP packets. Both of these data structures will be replicated, llinfo_arp [NBURBS] and arpintrq [NBURBS].

[0104] Sockets (struct socket)—The data structures for a socket are created dynamically for each socket, so no fixed replication needs to be done. The alteration of the protocol driver lookup code means that a socket will automatically get the pointer to the protocol driver in the correct burb without any changes to the socket code except the domain list search routines. A burb field will be added to the socket structure, socket.so_burb, so that it knows which burb it's in.

[0105] There are some other issues. A message doesn't normally have an intrinsic burb, but it will be added as a debugging aid during development. Each major data structure that is burb dependent (the input queues, the protosw structures, etc.) will always have a field giving its burb. So at various points, these fields can be compared against during development as a debugging aid. Critical hand off stages (like socket initialization) will have sanity checks that are done all the time because the overhead is negligible.

[0106] The kernel is implemented to respond to connection requests that are not intended for the computer system itself. For instance, when an internal client requests a connection to an external Internet host, the kernel will answer the request as if it were the external host. Meanwhile, it will attempt to set up the connection with the external host. This process occurs transparently to the internal client. Such support allows the secure computer to answer in a secure manner connections intended for outside boxes.

[0107] Each burb will have its own routing tables. The general algorithm for each incoming packet will be, see if it's to one of the local addresses, if so accept it (as this is standard behavior). If not, route it within the burb. If that fails, then see if there is a route in another burb (using other burbs' routing tables). If not, then toss the packet. If so, accept the packet and pass it up the stack, which eventually should arrive at the proxies.

[0108] The existing TCP and UJDP drivers will happily accept the packet and pass it onto a socket if it exists. If it's to an already established TCP connection or a UDP socket, it will get there normally. If it's a new incoming TCP connection that is accepted, it will create the socket, and fill in the local address from the incoming packet. The process doing the accept 0 can call getsockname 0 to get that address. In other words, the UDP and TCP drivers trust the lower layer 0 driver that the packet is for the machine, so no change to those layers need to be done at all to support this, only the secondary routing lookup in the IP layer.

[0109] This functionality means a couple of things. For this to work, there must still be unique IP addresses all around, so no hacks are done to support hidden IP addresses that conflict with external ones. Also, there can only be one default route on the machine. If there were one per burb, the burb routing lookup wouldn't know where to send it. This isn't hard. Generally the default route will be for one of the outside burbs. Other burbs will simply have to have complete routing for the internal network. This isn't a problem as it's really already the case. A host with a complex internal network already has to have proper routing for all of those networks if it uses a default route.

[0110] This also means there can be no duplicate routes in the replicated routing tables. If there were duplicate routes, the routing lookup would always return the first route found.

Network Separation Using Type Enforcement

[0111] The degree to which network separation protects a network from malicious attack is enhanced through the use of type enforcement.

[0112] As noted above, in a type enforcement scheme processes and files are assigned to a domain and subtype, domain:subtype. Process-to-file, or domain-to-domain:subtype, access is controlled by the Domain Definition Table (DDT), that specifies the permissions:

[0113] create—ddt_create

[0114] write—ddt_write

[0115] read—ddt_read

[0116] destroy—ddt-destroy

[0117] execute—ddt_execute

[0118] rename—ddt_rename Subtypes allow a finer division of file types (file file, diry=directory, sock=socket, exec=executable, etc.).

[0119] Interactions between the processes, or domain-to-domain, such as signals, are enforced by the Domain Interaction Table (DIT), that specifies the allowed signals (sHUP, sABRT, sjob, etc.) An instantiated domain is one that has several instances with identical access rights to itself and other domains. Take two instantiated domains, www1 and www1. They both have identical access to all other domains and subtypes. They also have equivalent access to their instances and no access to other instances. In the domain specification language, these domains are specified with 'X' as the last character, where X represents the burb ID. So if you say:

[0120] wwwX:

[0121] has read access to www#:conf

[0122] has write access to Slog:sock

[0123] the DDT/DIT is created like:

[0124] www0:

[0125] has ddt_read access to www0:conf

[0126] has ddt_write access to Slog:sock www1:

[0127] has ddt read access to www1:conf

[0128] has ddt write access to Slog:sock

[0129] If another domain is given access to a general instantiated type, then it gets equal access to all instances, for instance:

[0130] Admn:

[0131] has write, read, create access to www\#:conf

[0132] the DDT/DIT is created like:

[0133] Admn:

[0134] has ddt write, ddt read, ddtcreate access to www0:conf

[0135] has ddt_write, ddt-read, ddt_create access to www 1:conf

[0136] There are a set of special domains that are used to actually grant access to the networks themselves. These domains are in the form of Protocol:port. Protocol is a domain name corresponding to the protocol type. Current domains and what type of socket they map to include:

[0137] For sockets that have port numbers, such as tcp and udp, the subtype is an ASCI-ish number that is the port to which binding is allowed. For instance, the ftp0 domain would have access to ftp 0:0020 and ftp 0:0021 to give it access to the ftp control and data ports (**20** and **21**).

[0138] The access given corresponds to the directionality of communication. There must be ddt_create privilege to create the socket at all. For datagram sockets, ddt_read gives the ability to read 0 or recvmsg 0, and ddt_write gives access to write 0 and sendmsg 0.

[0139] With stream sockets, ddt_read gives access to bind 0, listen 0, and accept 0 so that the process can receive an incoming connection. Ddt_write gives access to bind 0 and connect 0 so that the process can initiate an outgoing connection.

[0140] The following subtypes give access to random and reserved ports:

[0141] ral—bind to all reserved ports explicitly

[0142] rany—bind to any arbitrary reserved ports

[0143] nall—bind to all non-reserved ports explicitly

[0144] nany—bind to any arbitrary non-reserved ports

[0145] For portless types, the subtype grants access to a specific type of socket. The types currently include 'icmp', only valid under the IP domain (ipoX:icmp), which gives access to ICMP sockets. Some more network subtypes:

[0146] sock—flag access to a network type

[0147] conf—access to network configuration capability

[0148] Subtype 'sock' is used during the socket creation so the kernel can verify access to a given network domain. The port isn't known at this point, and an exhaustive search of the DIT would be expensive, so any domain that has access to any networking domain has to be given access to the 'sock' subtype as well.

[0149] Network configuration programs such as ifconfig, nss, and sysctl must have access to "network_domain:conf". For example: tcpX:conf allows the Unix commands ifconfig to configure network interfaces and sysctl to set networking configurables.

[0150] A network interface is bound to a burb at boot time. A given socket is bound to a particular burb at the socket creation time. The rules for this are pretty simple.

[0151] An example of a network separated system **70** implemented with Type Enforcement™ protection is shown in **FIG. 5**. In **FIG. 5**, ftp0 and ftp1 are two instantiated domains (72.0 and 72.1, respectively). Each instantiated domain has access to its own protocol stack (**74.0** and **74.1**). Transfers from one domain to another are made through a ftp proxy **75** via calls to kernel **76**.

[0152] Processes running in an "instantiated domain" such as ftpo, intrinsically have access to one and only one burb, the burb that matches the numeric part of the domain name. In these cases, the system just binds that socket to the appropriate burb. It should be noted that a process is not actually bound to a burb until it tries to make a direct or indirect socket call to that burb.

[0153] Programs running within other domains either have no access or access to all burbs. In these cases, the caller must specify which burb they want. If they don't, it's an error. If they do specify a burb and they don't have access privileges to that burb, it's also an error.

[0154] This specification is implemented via a new system call, socketburb 0. Basically, socketburb 0 replaces the old socket 0 system call. It performs the same function and takes one additional argument, burbid. The old socket 0 still exists, but its gut has been replaced by a call to socketburb 0.

[0155] The syntax of such a call could be as follows: int socketburb (int domain, int type, int protocol, unsigned long burbid);

[0156] Ported applications and new programs will use the new socketburb 0. Existing programs, in instantiated domains, using socket 0 wouldn't have to be changed and still work. Burbification of socket 0 calls to socketburb 0 should use the library call domain_to_burb 0 to convert and pass the burbid argument:

[0157] sock1=socket (AF INET, SOCK_STREAM, 0);

[0158] sock2=socket (AF_INET, SOCK_RAW, IPPROTO_ICMP); would become:

[0159] int domain_to_burb (long domain);

[0160] sock1=socketburb (AP_INET, SOCK_STREAM, 0, domain_to_burb (domain));

[0161] sock2=socketburb (AF-INET, SOCK-RAW, IPPROTO_ICMP, domain to burb (domain));

[0162] Interfaces are bound to a particular burb. This is implemented by a few new socket ioctls0s which set and get the burb id of an interface.

[0163] Access to a socket for controlling interfaces and other shared parameters is controlled by access to the various conf subtypes. Parameters specific to a protocol, like TCP tunables, are controlled by access to the protocol specific type, such as 'tcp0:conf'. IP layer things like the IP address and burbness of an interface are controlled by the types like ip0:conf.

[0164] The following interfaces have been modified or added to support type enforcement, burbification, and transparent proxies. The objectives of these calls have not changed, although access to these calls has been made more restricted. Some new error codes, ermo, have been added in the case of type enforcement failures. Only the modifications are described below.

[0165] socket 0—is used by a process to create a socket that is bound to a particular burb.

[0166] If socket 0 is called, the socket is either bound implicitly to the burb id that matches the number of the instantiated domain, or fails.

[0167] socketburb 0—a new system call and it is used to create a socket in a given burb.

[0168] The burb file is passed as an additional argument to the socketburb 0 call:

[0169] int socket (int family, int type, int protocol);

[0170] int socketburb (int family, int type, int protocol, unsigned long burb);

[0171] int domain_to_burb (long domain);

[0172] sock1=socket (AF_INET, SOCK_STREAM, 0);

[0173] sock2=socketburb (AF_INET, SOCK_RAW, IPPROTO_ICMP, domain_to-burb (domain));

[0174] These new error codes, ermo, are returned by socket 0 and socketburb 0:

[0175] [EDBOM] returned and an audit generated if the calling process is not in a burb-bound domain.

[0176] [EPERM] returned and an audit generated if the calling process doesn't have ddt_create privilege.

[0177] bind 0—assigns a local address and port number to a socket. The bind (call has been extended to allow a process to set the local address arbitrarily, it needs ddt_rename access to "ipoX:spuf". Currently, no domains have this spoofing capability, i.e., no domains have ddt_rename access to "ipoX:spuf". The calling process also must have the following privileges to successfully bind:

[0178] ddt_read—tcpX:port

[0179] ddt_read—udpX:port

[0180] ddt_read—ipoX:port, for raw IP

[0181] has_rootness—reserved ports (0-1023)

[0182] [EADDRNOTAVAIL) returned and an audit generated if the calling process doesn't have ddt-rename privilege.

[0183] [EPERM] returned and an audit generated if the calling process doesn't have ddt_read or has_rootness privilege.

[0184] connect 0—initiates a connection from a socket to a specified address and port number. The calling process must have these privileges to the listed domains: subtypes to successfully connect.

[0185] ddt_write—tcpX:port

[0186] ddt_write—udpX:port

[0187] [EPERM] returned and an audit generated if the calling process doesn't have ddt_write privilege.

[0188] The ioctl 0 call is used to manipulate the underlying device parameters. Several new socket ioctl 0 commands have been added. Most will work with a "struct ifreq" argument and be handled similarly to the other set/get ioctls of an interface. If an ordinal value is being set or retrieved such as the burb id, then it will be passed in the ifr_metric field.

[0189] The following ioctl 0 commands now require is_startup or is_admin privilege to execute: SIOCSIFFLAGS, SIOCSIFMETRIC, SIOCADDMULTI, and SIOCDELMULTI. EPERM is returned and an audit is generated if the calling process doesn't have the correct privilege. These new ioctl 0 commands are added to support burbness in the secure computer:

[0190] SIOCGIFBURB/SIOCSIFBURB—get and set the burb ID of a network interface.

[0191] The burb ID is passed in the ifr_metric field. The CGI abbreviation in the first burb ID represents the ANSI/ISO computer graphics interface standard. To set the interface's burb ID, the calling process must have is_startup or is_admin privilege. A sample call would look like:

[0192] int ioctl (int socket_descriptor, unsigned long command, char *argp);

[0193] struct ifreq if;

[0194] int burb, so;

[0195] ifr.ifr_metric=burb;

[0196] of (ioctl (so, SIOCSIFBURB, (caddr_t) & ifr)<0 perror ("ioctl (SIOCSIFBURB)");

[0197] if (ioctl (so, SIOCGIFBURB, (caddr_t) & ifr)<0) perror ("ioctl (SIOCGIFBURB)");

[0198] [EPERM] returned and an audit generated if the calling process doesn't have is_startup or is_admin privilege.

[0199] [EINVAL] returned and an audit generated if the calling process doesn't have access to subtype ":conf".

[0200] SIOCGSOCKBURB—get the burb that a socket is bound to. This is mainly a debugging aid. The burb id is passed in the ifr_metric field.

[0201] SIOCGMSGBURB—get the burb of the pending message. It should always be the same as the

burb of the socket. This is mainly a debugging aid. The burb id is passed in the ifr_metric field. A sample call would look like:

[0202] struct ifreq ifr;

[0203] int burb, so;

[0204] if (ioctl (so, SIOCMSGBURB, (caddrt) & ifr) <0 perror ("ioctl (SIOCMSGBURB)");

[0205] burb ifr.ifr_metric;

[0206] [ENOENT] returned and an audit generated if the socket cannot be read.

[0207] SIOCGMSGIFNAME—get the name of the interface that the pending message arrived on. It is used as a debugging aid or to filter on the basis of the specific interface of the message. The interface name is returned in the ift name field. A sample call would look like:

[0208] struct ifreq ifr;

[0209] int so;

[0210] if (ioct (so, SIOCMSGIFNAME, (caddr_t & ifr)<0 perror ("ioctl (SIOCMSGIFNAME)");

[0211] printf ("interface: %s", ifr.ifr_name);

[0212] [ENOENT] returned and an audit generated if the socket cannot be read.

[0213] SIOGSUBURBRT—returns a burb id for a given IP addresses. The kernel searches each routing table and returns the first route found. The IP address is passed in the ifr_addr field. The burb id is returned in the ifr_metric field on success. Since there can be more than one burb, this command is used by an incoming proxy to find which outing proxy to connect to. A sample call would look like:

[0214] struct ifreq ifr,

[0215] int so;

[0216] ifr.ifr_addr=tempaddr;

[0217] if (ioctl (so, SIOCGBURBRT, (caddr_t) & ifr)<0) perror ("ioctl (SIOCGBURBRT)");

[0218] printf ("route %d", ifr.ifr_metric);

[0219] [EHOSTUNREACH] returned and an audit generated if no route can be found for the given address.

[0220] The getsockopt 0 and set sockopt 0 calls manipulate the options associated with a socket. The following socket options have been added:

[0221] #include <sys/types.h>

[0222] #include <sys/socket.h>

[0223] int getsockopt (int so, int level, int optname, void *optval, int *optlen);

[0224] int setsockopt (int so, int level, int optname, const void *optval, int optlen);

[0225] SO_STATE—returns the state of the socket, the so_state field of the struc socket. This is used by applications that wish to be more intelligent in their handling of a socket depending on its state.

[0226] SO_SOCKBURB—returns the burb id of the socket, similar to ioctl (SIOCGSOCKBURB). A sample call would look like:

[0227] int value, length, error;

[0228] length=size of (value);

[0229] if ((error=getsockopt (sockfd, SOL_SOCKET, SO_SOCKBURB, & value, & length return error;

[0230] return value;

[0231] SO_MSGBURB—returns the burb id of the pending message, similar to ioctl (SIOCGMSGBURB) via an unsigned long.

[0232] [ENOENT] returned and an audit generated if the socket cannot be read.

[0233] SO_MSGIFNAME—returns the burb id of the pending message, similar to ioctl (SIOCGMSGIFNAME) via an unsigned long.

[0234] [ENOENT] returned and an audit generated if the socket cannot be read.

[0235] Most of the kernel auditings will be generated by type enforcement checks such as check_ddt 0, check_dit 0, and check_ddt_net 0 and user privilege check such as psuser 0. These audits are logged to /var/log/audit.asc and /var/log/audit.te.asc.

[0236] Network audit is another type of kernel audit. These are logged to /var/log/audit.asc and /var/log/audit.attack.asc. Currently, network audits are generated for ICMP, TCP, UDP, and IP packets.

[0237] ICMP—ICMP messages are used to communicate error and administrative messages between systems. ICMP audits are the only network audits that can be configurable. This is implemented via the Unix command sysctl and the field burb.X.net.inet.ip.icmpaudit, where X is the burb id. The default state of the secure computer is level 1, audit only icmp-redirects. ICMP audits can be configured at 3 levels:

[0238] 0 no icmp audits

[0239] 1 icmp redirects only (default)

[0240] 2 all icmp messages, except echo and reply

[0241] IP—if the source-routed packets option is disabled (0), the kernel will generate a "source-routed packets dropped" audit for each source-routed packet. The source-routed packet option is configurable via the sysctl command, by disabling or enabling the fields burb.X.net.inet.ip.forwarding and burb.X.net.inet.ip.forwsrcrt. As default, both of these fields are disabled so the secure computer will not forward any source-routed packets.

[0242] TCP—the kernel will generate an audit for any TCP attempts to synchronize sequence numbers without a valid protocol control block. These types of packets are viewed as hostile probing attempts.

[0243] UDP—similar to TCP, the kernel will also generate a probing attempt audit for any UDP attempts to connect without a valid protocol control block.

[0244] There are a number of system utilities that configure or retrieve the network states. These utilities are also modified to support burbness.

[0245] arp—displays and modifies the Internet-to-Ethemet address resolution. The arp display and command line interface will remain the same. Only the internal implementation will be modified to handle the replicated arp tables.

[0246] ifconfig—is used to configure network interface parameters. It is modified to accept a "burb id" entry so that it can be used to set the burb of an interface. It will also print the burb of an interface as part of its normal output. Example: if config ef0 inet 172.17.128.79 netmask 255.255.0.0 burb 0.

[0247] inetd—is the internet-service daemon that runs at boot time and listens for connections on certain internet sockets. This daemon will be replaced by the Network Services Sentry, NSS. NSS will be "burb aware" and able to start network services in the appropriate domain. Resident servers will be launched from a similar master utility, such as /etc/rc, so that they run in the correct burb.

[0248] netstat—formats and displays various network-related status and data structures. The formatting has been modified so that when appropriate the display includes a burb ID column in its output.

[0249] sysctl—is used to retrieve kernel state and allows processes with appropriate privilege to set kernel state. Network states are replicated appropriately into NBURBS.

[0250] All services supported on the secure computer, such as telnet and ftp, pass their data through a set of proxies (see, e.g. ftp proxy **75** in **FIG. 5**). There is one proxy for each type of service, http proxy for web, telnet proxy for telnet, and so on. The proxy will accept 0 the connection or rcvmsg 0/recvmsg 0 the packet and look at the incoming address, either in the mesg structure for UDP or via getsockname 0 for TCP. It will then find out which burb has a route to that destination, via ioctl (SIOCGSUBURBRT), and establish a socket connection in the outgoing burb.

[0251] That connection will be via a Unix domain socket. Each service will have a proxy directory in Ivar/run, such as /var/runitelnetp/. In that directory each outgoing proxy will have already opened a domain socket. These sockets exist in the file namespace, so they already have access control via the domain and type of the socket node. Thus, the burb functionality guarantees that a given process can only access one burb, and the type enforcement on the socket guarantees that process can't cheat and access another service's proxies. The types on the sockets can give a finer control within a service if needed.

[0252] So the following example shows a telnet proxy servicing a telnet from the external network Net1, into the internal network, Net0. /etc/rc is the boot up script, telnetd is the telnet daemon, and nss is the Network Services Sentry daemon.

	read()s destination does a connect () to the real destination passes back success
select () loop passing data	select () loop passing data
cleanup on close	cleanup on close

[0253] Packets can be filtered as part of the process. Filtering could go before the proxy, it could go after the proxy, or it could go inside the proxy.

[0254] Some things, like fip, are more complex. The proxies have to watch for PORT and PASV requests and set up the data channel and proxy it. Additional channels can be either set up in advance with a standard protocol, or created on the fly by having proxies set up a control channel and bind additional separate domain sockets.

[0255] UDP proxies are also more complex. The proxies may have to maintain state information about requests, so that they know which request goes where. A UDP (or other datagram) proxy can work two ways. It can either have a single proxy per burb that just forwards and filters packets one at a time (in which case the connecting Domain socket is a datagram socket as well), or it can create a proxy for each "pseudo" connection and track state to optimize things (like sending acks, etc.). In this case there could be a pair of proxies on the fly that created a stream connection between themselves. They could be routed by port (for protocols that have a unique port and each end negotiated on the fly, like gopher or DNS), otherwise a single proxy still has to handle incoming packets and dispatch them.

[0256] Which approach is to be used depends on a few things. If address hiding and masquerading is done, state must be maintained. If not, then simple packet forwarding can be done. Some protocols, like NFS (if ever done) will require extensive state. Further information on network separation can be found in "SYSTEM AND METHOD FOR ACHIEVING NETWORK SEPARATION", U.S. patent application Ser. No. 08/599,232 by Gooderum et al., filed Feb. 9, 1996, the details of which are hereby incorporated by reference.

Type Enforcement Protection of Compiled Program Code

[0257] Type enforcement can be used to extend the access protections inherent to a particular program, without access to the source code for that program. An overview of the process used to install any pre-compiled application binary (i.e. the program) onto the secure computer system is shown in **FIG. 6**. The first step (100) is to install the binary to a location from which it can be executed. Some (but not all) examples of how this can be accomplished are: (a) copying from the installation floppy disk(s) to the internal hard drive, (b) copying from the installation cdrom to the internal hard drive, (c) mounting external media containing the application binary, and (d) copying the application binary via the network to the internal hard drive.

[0258] Once the application binary is accessible for execution, the secure computer system is placed in a different state than the fully secure operational state. This state is called the development state, since in best practice (i.e. for highest security) this state is only available to developers. The

development state has all of the type enforcement checks that the operational state has; however, the consequences of a violation are different. In operational state, a type enforcement violation results in denying the requested access. In development state, a type enforcement violation results in allowing the requested access. In both the operational and development state, a log entry is produced which records the type enforcement violation. This logging capability is needed to determine which type enforcement violations the program is generating.

[0259] It should be noted that the system does not have to be placed in development state to perform this procedure. All that is absolutely required is the logging capability. However, by placing the system in development state, the time and effort required to perform this procedure is greatly reduced. In practice, the procedure detailed in **FIG. 6** is followed under in the operational state only when an error was made when performing the procedure under the development state.

[0260] At step 102 the program is executed. Because this program was developed using non-type enforcement techniques, this program will make assumptions about its execution environment that will not be true under the type enforcement execution environment. These faulty assumptions will manifest themselves as type enforcement violations. Some sample faulty assumptions that the program can make include (but is not limited to): (a) the program can create files in the system-wide temporary file area, (b) the program can read files that have Unix "global-read" permissions set to true, etc. These faulty assumptions are a direct consequence of the program being non-type enforcement aware: the program is unaware that, in addition to the Unix security restrictions that it is expecting, there are type enforcement security restrictions. Every type enforcement violation must be successfully dealt with in order for the program to function properly while under the operational state.

[0261] There are three general ways to successfully deal with a type enforcement violation. They are: (1) ignore the violation, (2) remove the request that generated the violation, (3) grant additional permission(s) to remove the violation. The criteria for choosing one branch over another branch are functionality and security. The less critical the functionality, the more likely that the violation can be ignored or the request removed. The more critical the functionality, the more likely that additional permission(s) will be granted.

[0262] Some violations can be safely ignored. For example, some programs attempt to determine how busy the system is, in order to behave differently under different conditions. If, however, the program is unable to determine how busy the system is (because of a type enforcement violation), the program will simply assume that the system is not busy and continue. For some programs this is an acceptable response, and requires no further action; in these cases (**108**), type enforcement violations can be ignored.

[0263] Some violations can be removed by changing the environment that the program is running under. For example, some programs attempt to change their effective user identification after some resource has been acquired. The most common situation of this kind is when the program runs as "root" long enough to open a file, but then changes to "wwwdaemon" immediately thereafter. Under type enforcement, this ability to change to another user will result in a violation under the default operational state. In cases such as these, it is more security effective to remove the need for the program to change to another user by following these steps:

[0264] (1) start the process as user "wwwdaemon" directly (thus, the program will no longer require the ability to change to another user);

[0265] (2) set the Unix permissions on the file such that it can be read by user "wwwdaemon"; and

[0266] (3) set the type enforcement permissions on the file such that it can be read by the domain in which the process started in step (1) is executing.

[0267] So, in this example, the original type enforcement violation is removed at 110 by changing the environment external to the program. Note that in a non-type enforcement system, step (2) results in a security vulnerability. It is only through step (3) that the security vulnerability is closed.

[0268] In the above example, one can think of the original type enforcement violation as being transformed into another type enforcement violation. The ability to "change user" was transformed into the ability to "read a file".

[0269] Some violations can be removed by granting the program appropriate type enforcement permission(s), so that what was a violation now is a granted permission. Once it is determined that the violation can neither be ignored nor transformed, this action is performed (112). There are three general kinds of type enforcement violations. They are: (1) File Access, (2) Process-to-Process Communication, and (3) System Environment Access. Therefore, a check must be made at 114 to determine the variety of type enforcement violation.

[0270] If the violation concerns general file access, a check is made at 116 to see if the file is a shared file. Programs typically need access to many different sorts of files: configuration files, temporary or scratch files, permanent data files, etc. The only factor that distinguishes these sorts of files from another in the type enforcement sense is whether or not the file is a shared file or not. An example of a non-shared file could be a temporary file. Here, only the program needs access, and no other. An example of a non-shared file could be a database file containing a list of every user of the system.

[0271] If the file is shared, then a shared type must be used. Sometimes, this shared type will not yet exist, and must be created. This can occur, for instance, when two new programs both access the same new file (for example, a configuration file that one program writes and the other program reads). Sometimes, the shared file will already exist. This can occur, for instance, when the new program accesses a system-wide file (for example, a database file containing a list of every user of the system.) If the shared file already exists, it will already have a type. This type is used at 118.

[0272] If the file is not shared, a new type must be created, and the file is set to this new type.

[0273] In either case (both shared and non-shared), the final step is to add at 120 a direct permission to the type

enforcement database. This type enforcement permission will grant the domain (in which the program is running) permission on the type (and the file is then set to that type). The fact that the file is not shared is manifested in that there will not be another domain with permission on the type. The fact that the file is shared is manifested in that there will be another domain with permission on the type.

[0274] The second kind of violation concerns process-to-process communication. Some programs require the ability to communicate with another program via a process-level mechanism. To deal with this kind of violation, one determines the domain of the second program (the program that the first program is trying to communicate with). Then, at 122 a direct permission is added to the type enforcement database. This permission will grant the sending domain (in which the first program is running) permission on the receiving domain (in which the second program is running). Permissions of this kind are called domain-to-domain interactions, and effectively control the permitted lines of communication between processes.

[0275] The third kind of violation concerns system environment access. This kind captures all the kinds of violations that do not concern file access or process-to-process communication. An example would be the ability to determine how busy the system is. Previously, this functionality was used to demonstrate a case where the ability could be removed. However, if the ability was required, then dealing with the type enforcement violation would be handled under this kind of violation. To deal with this kind of violation, one determines the specific functionality required. Then, at 124 a direct permission is added to the type enforcement database. This permission will grant the program's domain permission to invoke the required system function.

The Use of Both Type Enforcement and Network Separation to Protect Compiled Program Code

[0276] The process described in the previous section is sufficient to handle those instances where programs do not have to be compartmentalized to achieve sufficient security (e.g. utilities such as screen savers or logging utilities). In that case, all programs can reside in and be bound to one region or burb. For systems which can be partitioned into distinct burbs, however, an additional degree of protection can be added by incorporating network separation.

[0277] In the case, for instance, where the program(s) to be installed can reside in multiple burbs, yet must maintain the ability to communicate with each other via a networking interface, network separation is used to increase the overall security of the system. One example of such a case is where the system consists of two programs: a production-side program and an administration-side program. In these instances, it is required that the production-side program reside in an unprotected (e.g. Internet) burb. But one does not want the administration-side program residing in an unprotected (e.g. Internet) burb. So instead, the administration-side program is placed in a protected (e.g. internal) burb.

[0278] However, by splitting the two programs (production and administration) into two burbs, one has again changed the actual execution environment away from the expected execution environment. The two programs expect (and rely on) the execution environment to allow process to process communication (e.g. signals) between the two programs. On a non-Network Separated system without Type Enforcement™ protection, this assumption is valid. However, on a Network Separated system, the act of placing the administration-side program in a different burb than the production-side program will have the Network Separation side effect of disallowing process to process communication between the administration-side program and the production-side program. From a functionality standpoint, this is not acceptable.

[0279] To restore the required program to program communication functionality, one is required to perform steps outside of those explained in the previous section. The problem to be solved concerns the conflicting restrictions placed by the type enforcement system and the network separation system.

[0280] In Network Separation, if one program has been "bound" to one burb (e.g. Internet) and another program has been "bound" to another burb (e.g. internal), the Network Separation mechanism will preclude the ability for the first program to establish a connection to the second program via a network communication connection; thus it precludes the ability for program to program communication using the network. One way of viewing Network Separation having Type Enforcement™ protection is to acknowledge this lack of direct network to network communication: since the networks cannot be shared, they are separated.

[0281] So, the problem here can be restated as follows: given a program bound to one burb and another program bound to a different burb, how can we allow the two programs to communicate with each other without having access to the source code for either program.

[0282] The procedure to solve this problem is shown in **FIG. 7**:

[0283] First, at 140 decide to which burb each of the separate programs is to be bound. In the case of a production-side program and an administration-side program, it is clear that the production-side program must be bound to an Internet burb, and it is clear from a security standpoint that the administration-side program should be bound to an internal burb. In the case of more complicated sets of programs, tradeoffs must be made between availability and security. Binding a program to an Internet burb increases availability but also enormously increases security risks. Binding a program to an internal burb decreases security risks, but results in a decrease in availability.

[0284] Second, at 142 determine how the two programs interact. Again, it is important to note that source code is not available for this step. If source code is available for inspection and modification, then this procedure does not apply. Without source code, this step is accomplished by examining the type enforcement violations generated by either or both programs, as required.

[0285] Third, at 144 determine whether the type enforcement and network separation violations discovered in the second step can be eliminated. Techniques for removing TE violations (such as those shown in **FIG. 6**) can be used. In addition, one may

be able to eliminate a type enforcement violation by placing one of the programs into a situation where it is "partially bound" to a particular burb. A "partially bound" program is a program which can interact with a specific burb under a reduced set of Network Separation rules. That is, a partially bound program can perform process to process communication with another burb but not network to network communication. Specifically, it is possible for the partially bound program in a first burb to signal a program that has been completely bound to another, separate burb, while simultaneously maintaining other connections to the first burb. Therefore, communication between programs assigned to different burbs is permitted in a limited way and the problem is solved.

[0286] In one embodiment, one creates this "partially bound" program by using a type enforcement database trick. Normally, a program "P" is completely bound to a burb when the following syntax is used:

[0287] "P" runs in domain www#

[0288] which means that program "P" can access any of the burbs 0 through 9 (as signified by the character '#' above). When program "P" first starts running, it has its choice of burbs (e.g. www0, www1 . . . www9) to establish a connection. But after the first connection, the program "P" is fully bound to that burb and the domain that is associated with that burb, and cannot establish a connection to any network in another burb.

[0289] A program "P" is "partially bound" to a burb when the following syntax is used:

[0290] "P" runs in domain adm0

[0291] Here, the program "P"**0** loses some of its flexibility (it can now connect only to burb adm0) but it gains the "partially bound" status. Furthermore, since such an approach is treated as if there are no other instantiated domains (i.e. no adm1, adm2, etc.), program P running in domain adm0 can now connect to all other "#"-defined domains.

[0292] For the production-side program "PP" and administration-side program "AA" example above, a full example might be:

[0293] "AA" runs in domain adm0

[0294] "PP"runs in domain prd#

[0295] Thus, since the program "AA" is "partially bound" to 'adm0', there can be no domains adm1 through adm9. Program "AA" has, however, gained the ability to interact with prd0 through prd9. Thus, the required signaling ability is reestablished.

[0296] If it is not possible to place all but one of the programs in a "partially bound" burb, then this procedure must stop without succeeding. The only alternative at this point is to place all programs in the same burb. Then, since the additional Network Separation checks are no longer an issue, this procedure degenerates into the type enforcement only approach described previously.

[0297] Further information on increasing security of compiled program code can be found in "SYSTEM AND METHOD FOR SECURING COMPILED PROGRAM CODE", U.S. Pat. No. 5,867,647, by Haigh et al., issued Feb. 2, 1999, the details of which are hereby incorporated by reference.

Secure Commerce Server

[0298] One embodiment of a binary program made secure through a combination of type enforcement and network separation is described next. The Netscape Commerce Server, version 1.1, available from Netscape Computer, Inc., is a server for use in conducting commercial transactions over the Internet. This server provides user authentication, SSL protection to http connections, and a forms interface for server administration. As such, it is critical that care be taken to protect a malicious attacker from gaining access to and subverting the program.

[0299] A network separated embodiment **160** having Type Enforcement™ protection of the Netscape Commerce Server system is shown in **FIG. 8**. The Netscape Commerce Server system includes two distinct servers: the Commerce server **162** and the administration server **164**. In the preferred embodiment, one administration server **164** should be able to administer and configure a plurality of separate Commerce servers **162**. In the system of **FIG. 8**, the dashed lines indicate a user level permission check such as the Unix file permissions. In addition, components of the system are placed within domains **166**, **168** and **172** for a higher level of security.

[0300] Commerce server **162** is installed via an HTML forms driven interface. Administration and configuration after installation is done in the same manner. Installation operates as follows:

[0301] 1. The user runs the script ns-setup in the source tree (the directory structure from which files are copied to install the Netscape Commerce Server).

[0302] 2. The user is prompted for the hostname of the host machine.

[0303] 3. A temporary configuration is written in \tmp and a special WWW server is started using this configuration (the installation server).

[0304] 4. The installation server binds to an arbitrary, non-reserved port.

[0305] 5. The user is prompted for the name of a Web browser to start up. It then starts the browser on the port the installation server is listening on.

[0306] 6. The rest of the installation is HTML forms-driven through the browser.

[0307] Various items such as port number for the Commerce server, UID to run the server under, install directory, logging, administration password, and other server configuration are entered via three forms.

[0308] 7. The configuration is verified and the actual installation takes place:

[0309] 1. create destination directory and subdirectories

[0310] 2. create document root directory

[0311] 3. create server startup and shutdown scripts

[0312] 4. create server configuration files

[0313] 5. copy all binaries, administration forms, and other files from source tree to destination tree

[0314] 6. change owner of directories

[0315] 7. start administration server

[0316] 8. start Commerce Server

[0317] 9. remove files created in /tmp

[0318] 8. Installation complete.

[0319] Many Commerce servers **162** can be run on the same machine, binding to different ports (or even the same port, using different IP addresses) and having their own configuration. Each server **162** preforks **16** processes (number is configurable) to serve requests. This can load the system down if many servers are running.

[0320] After installation, Commerce server **162** is administered via forms using any forms-capable browser and connecting to a separate administration server running on its own port. This server is started and stopped manually, except during installation when it is started automatically. (It is recommended by Netscape that it be stopped when not being used.) The administration server is also configurable via forms (it configures itself). It can be configured to require a password to access it, and to allow only certain hostnames or IP addresses to connect to it.

[0321] The Commerce server **162**, once running, serves Web pages from a directory tree whose root is configurable. CGI script location, URL mappings, user authentication, access control by host, logging, and other items are also configurable via the administration server. Configuring some items causes creation of files, such as generating a key, installing a certificate, and creating a user database.

High Level Design

[0322] This section describes what must be done to port the Netscape Commerce Server to the secure computer. Since we have no source code from Netscape, we cannot modify how the server or installation process operates. Thus, design will focus on what needs to be done outside the Netscape source code to handle type enforcement.

[0323] The installation process will be done completely in the administrative kernel (no type enforcement checks), so setting file types in the source tree is not necessary. All file types must be set after the install process is complete since Netscape is not type-enforcement (TE) aware. To accomplish this, a script is used to set all types appropriately. The script must be able to find the appropriate directories, because paths are configurable, and the server files are in a directory named for the port it binds to. This info is passed via arguments or entered by the installer via prompts.

[0324] Commerce Server **162** runs in domain **166**. In one embodiment, this is the same domain in which the CERN server (not shown) runs. Domain **166** is a bound (burbed) domain which is extended to handle the Commerce Server **162**. That is, the Commerce Server is extended to be able to bind to port **443** (https), **80** (http), **8000**, **8001**, and **8080**. In addition, to allow site flexibility, Commerce Server **162** must be able to bind to selected reserved port ranges.

[0325] Administration Server **164** runs in a new Netscape Admin domain, domain **168**. Domain **168** must be a burbed domain, since it does a non-burb-aware socket 0 call. Transfers between domains must be done through proxy programs acting in concert with kernel **170**.

[0326] CGI scripts run in a CGI processor **171** in burbed CGI domain **172**. Since, however, the Netscape server cannot be modified to do a makedomain 0 to run the CGI script in the proper domain, there will need to be a :tran type added to CGI domain **172**. Then, all CGI scripts will be of type tran. In an alternate embodiment, a line could be added to the CGI script itself which causes the script to automatically transition into CGI domain **172**. To run a CGI script, Commerce Server **162** initiates a new process which executes in CGI processor **171** within CGI domain **172**. The results are then transferred back to Commerce Server **162**. In one embodiment, the CERN server also executes CGI scripts within CGI domain **172**.

[0327] For installing two servers on the same port, but with different IP addresses:

[0328] 1. Follow the install directions twice, specifying everything the same except a different server name (which will need to resolve to the correct IP address), bind address, and document root.

[0329] 2. Make an IP alias so that both IP addresses used for the bind addresses in the previous step access the secure computer. Type: ifconfig efl alias 172.17.128.199 to alias the IP address 172.17.128.199 to the external interface.

[0330] 3. Access both servers using the different IP addresses.

Files Manifest

[0331] The following are files associated with the Commerce Server. In this section, /server-root/ will denote the path where the server files are installed. Also, the subdirectory https-**443** will denote the directory containing all files specific to the server running on port **443**. The actual name will contain the port number the server binds to in place of '443'.

[0332] Executables

[0333] /server-root/bin/https/ns-httpd

[0334] The Commerce server.

[0335] /server-root/admserv/ns-admin

[0336] The administration server.

[0337] /server-root/admserv/servlist

[0338] Part of the administration server.

[0339] /server-root/bin/https/admin/bin/*

[0340] CGI scripts for the admin server that generate forms, interpret responses, change config files, administer servers, generate keys, etc.

[0341] /server-oot/extras/database/batchdb

[0342] Utility to convert NCSA-style user database to DBM database.

[0343] /server-root/extras/database/changepw.cgi

[0344] CGI script to allow users to change their own password via forms.

[0345] /server-root/extras/log_anly/analyze

[0346] Analyze access logs.

[0347] /server-root/extras/log_anly/a_form.cgi

[0348] CGI script to analyze access logs via forms.

[0349] /server-root/https-443/restart

[0350] Restart the Commerce server.

[0351] /server-root/https-443/rotate

[0352] Rotate the log files.

[0353] /server-root/https-443/start

[0354] Start up the Commerce server.

[0355] /server-root/https-443/stop

[0356] Shut down the Commerce server.

[0357] /server-root/start-admin

[0358] Start up the administration server.

[0359] /server-root/stop-admin

[0360] Shut down the administration server.

[0361] Configuration files

[0362] /server-root/admserv/admpw

[0363] User and password file for access to administration server.

[0364] /server-root/admserv/ns-admin.conf

[0365] Configuration file for the administration server.

[0366] /server-root/bin/https/admin/html/*

[0367] Forms templates for the administration server.

[0368] /server-rootibinlhttps/admin/icons/*

[0369] Icons for the administration server.

[0370] /server-rootlmc-icons/*

[0371] Icons for the Commerce server, used for gopher and ftp listings.

[0372] /server-root/userdb/*

[0373] User databases. (Initially empty)

[0374] /server-root/https-443/config/admin.conf

[0375] Commerce server configuration file.

[0376] /server-root/https-443/config/magnus.conf

[0377] Commerce server configuration file.

[0378] /server-root/https-443/config/obj.conf

[0379] Commerce server configuration file.

[0380] /server-root/https-443/config/mime.types

[0381] Commerce server configuration file.

[0382] /server-root/https-443/config/ServerKey.der

[0383] Key pair file. Initially non-existent, generated by admin server when requested by the administrator. Path is configurable.

[0384] /server-root/https-443/config/ServerCert.der

[0385] Certificate file. Initially non-existent, installed by admin server from the Certificate Authoritrys certificate response when requested by the administrator. Path is configurable.

[0386] Log files

[0387] /server-root/admserv/errors

[0388] Log file for administration server.

[0389] /server-root/https-443/logs/access

[0390] Commerce server access log. Path is configurable.

[0391] /server-root/https-443/logs/errors

[0392] Commerce server error log.

[0393] /server-root/https-443/logs/secure

[0394] Commerce server secure log. (All https accesses are logged here with the keysize used).

[0395] Temporary files

[0396] /server-root/admserv/pid

[0397] Contains the process ID of the administration server when running.

[0398] /server-root/https-443/logs/pid

[0399] Contains the process ID of the Commerce server when running.

[0400] System files

[0401] /etc/spwd.db

[0402] System password file containing encrypted passwords.

[0403] /etc/pwd.db

[0404] A shadow password file.

[0405] Other files

[0406] /server-rootlextras/database/changepw.htm

[0407] HTML form to allow users to change their own password.

[0408] /server-root/extras/log_anly/a_form.html

[0409] HTML form to analyze access logs.

[0410] /server-root/docs/index. html

[0411] Default home page for the Commerce server. Path is configurable.

[0412] CGI scripts

[0413] Path is configurable.

[0414] HTML pages

[0415] Path is configurable.

[0416] As noted previously, Netscape Commerce Web server **162** runs in the Web server domain (domain **166**), the same domain that the CERN Web server runs in. Therefore, the Web server domain needs execute permission to the following:

[0417] /server-root/bin/https/ns-httpd

[0418] In addition, the Web server domain needs read permission to these files:

[0419] /server-root/mc-icons/*

[0420] /server-root/userdb/*

[0421] /server-root/ttps443/config/admin.conf

[0422] /server-root/https-443/config/magnus.conf

[0423] /server-root/https-443/config/obj .conf

[0424] /server-root/https-443/config/mime.types

[0425] /server-root/https-443/config/ServerKey.der

[0426] /server-root/https-443/config/ServerCert.der

[0427] /server-rootlextras/database/changepw.htm

[0428] /server-root/extras/log_anly/a_form.html

[0429] /server-root/docs/index.html

[0430] The Web server domain needs write permission to these files:

[0431] /server-root/https-443/logs/access

[0432] /server-root/https-443/logs/errors

[0433] /server-root/https-443/logs/secure

[0434] /server-root/https-443/logs pid

[0435] Since the server normally runs as root, if it were able to execute arbitrary code, this would be bad. We address this by not having it run as root, taking advantage of Type Enforcement™ protection of the sockets. This allows users other than root to connect to low numbered sockets. Other than needing to bind to port **80** or **443**, there is no other reason the server needs to run as root.

[0436] Therefore, the server will run as user "www" and group "www" with no need to run as root.

[0437] The CGI bin executables present a potential security concern. Since the Commerce server is allowed to execute these files, if someone were able to put their own executable on the secure computer, the server may be compromised. TE only allows the Web server domain to transition to the CGI domains and no others. It does not have permission to create or write to files of any executable type. This helps prevent the possibility of an external user subverting the server and uploading an executable file.

[0438] The entire directory tree containing the html documents which the Commerce server accesses will be write protected from the Web server domain. A separate writable directory will be set aside for all the data created by external users. This is where the CGI executables will put their results. The CGI domain will be allowed to create files in this directory, but not to read or destroy them. These files may later be read and moved internally by the webmaster using mail or FTP.

[0439] The password administration program will run in the Netscape Admin domain (domain **168**). This domain will have all the accesses required to maintain the Admin server and all Commerce servers on the secure computer.

[0440] The Admin domain needs execute access to the following:

[0441] /server-rootladmserv/ns-admin

[0442] /server-root/admserv/servlist

[0443] /server-root/bin/https/admin/bin/*

[0444] /server-root/extras/database/batchdb

[0445] /server-roottextras/log_anly/a_form.cgi

[0446] /server-root/extras/log_anly/analyze

[0447] /server-roothttps-443/restart

[0448] /server-rootthttps-443/rotate

[0449] /server-root/https-443/start

[0450] /server-root/https-443/stop

[0451] /server-root/start-admin

[0452] /server-root/stop-admin

[0453] The Admin domain needs read access for:

[0454] /server-root/bin/https/admin/html/*

[0455] /server-root/bin/https/admin/icons/*

[0456] /server-root/extras/database/changepw.htm

[0457] /server-root/extras/log_anly/a_form.html

[0458] HTML pages

[0459] The Admin domain needs both read and write access for:

[0460] /server-root/admserv/admpw

[0461] /server-root/admserv/ns-admin.conf

[0462] /server-root/admserv/pid

[0463] /server-root/admserv/errors

[0464] /server-root/userdb/*

[0465] /server-root/https-443/config/admin.conf

[0466] /server-root/https-443/config/magnus.conf

[0467] /server-root/lhttps-443/config/obj.conf

[0468] /server-root/https-443/config/mime.types

[0469] /server-root/https-443/config/ServerKey.der

[0470] /server-root/https443/config/ServerCert.der

[0471] /server-root/https443/logs/access

[0472] /server-root/https-443/logs/errors

[0473] /server-root/https-443/logs/secure

[0474] /server-root/https-443/logs/pid

[0475] CGI scripts run in CGI domain **172**. The following files will be of cgix:tran type and will run in the CGI domain:

[0476] /server-root/extras/database/changepw.cgi

[0477] /server-rootlextras/log_anly/a_form.cgi

[0478] CGI-scripts

[0479] Commerce Server **162** is controlled by the various configuration files maintained by administration server **164**. In one embodiment, it may be advantageous to add the ability to start and stop administrative server **164** from another system administration program running in another domain.

[0480] Since the secure computer uses type enforcement on sockets, we can allow users other than root to bind to reserved ports. As a result, we run the Commerce server as user id "www" and group id "www". The "www" user is

similar to "nobody" and has no log in capabilities. The administration server will still run as root. In addition, scripts that start and stop the servers need to execute /bin/sh (type $Sys:shel). These scripts are also not type-aware, so they need to be modified to execute in the correct domain.

[0481] In one embodiment, the following domains are used to define the commerce server system. It should be apparent that other combinations of domains, privileges or access rights could be used. In the list, each domain can have the following privileges: "is_admin" indicates the domain is an administrative domain, "has_rootness" means that the domain can violate Unix permissions if the process UID is root, "can_setlogin" indicates that the domain can set the login name of a process. In addition, each domain's DIT could have the following permissions: "dt" indicates that transition to the named domain is allowed, "sABRT" indicates that sending an ABRT signal to the named domain is allowed, "sjob" indicates that sending a job control signal to the named domain is allowed, "sHUP" indicates that sending a HIUP signal to the named domain is allowed, and "sUser" indicates that sending a USR1 or USR2 signal to the named domain is allowed.

[0482] Finally, each DDT uses the following permissions: "w" is permission to write, "r"]is permission to read, "d" is permission to destroy, "n" is permission to rename, "c" is permission to create and "e" is pernission to execute.

```
Domain: CGI 'cgix'
        { CGI scripts run here - limited access to things }
        Privs:
        DIT:
            CGI    sABRT      { }
        Subtype: Script 'scrp'
            { Used for CGI shell scripts }
            DDT:
                CGI    er    { }
                Webmaster    w r d n c      { }
                www er    { }
        Subtype: Transition 'tran'
            { Used for CGI compiled executables, not shell scripts. }
            DDT:
                Webmaster r e w n { }
                www e    { }
        Subtype: File 'file'
            { }
            DDT:
                CGI    w r d n c      { }
                Webmaster    w r d n c      { }
        Subtype: Directory 'diry'
            { }
            DDT:
                CGI    w r      { }
                Webmaster    w r d n c      { }
                www      { }
        Subtype: Save 'save'
            { }
            DDT:
                CGI    c      { }
                Webmaster    w r d n c      { }
Domain: NetscapeAdmServ 'nas0'
        { This domain is for the Netscape Admin Server. }
        Privs:
            can_setlogin
        DIT:
            CGI    dt         { }
            NetscapeAdmServ    sJob sABRT { To control itself. }
            www    dt sJob      sHUP { }
        Subtype: Executable 'exec'
            { }
```

-continued

```
            DDT:
                NetscapeAdmServ    r e    { }
                Webmaster    r e    { }
        Subtype: Config 'conf'
            { The Netscape Admin Server configuration file }
            DDT:
                Webmaster    r w n c d      { }
                NetscapeAdmServ    r w d n c      { }
        Subtype: File 'file'
            { Default file type. This is the type created
            whenever the Admin
            Server creates a file (Key/Certificate files,
            User Databases, temporary files, etc.). }
            DDT:
                Webmaster    w r d n c      { }
                NetscapeAdmServ    w r d n c      { }
                www r        { To read Key and
                               Certificate files, and User
                               databases created by the
                               Netscape admin server. }
        Subtype: Directory 'diry'
            { }
            DDT:
                NetscapeAdmServ    w r d n c      { }
                Webmaster    w r d n c      { }
Domain: Webmaster 'Webr'
        { This domain is for web admin work. }
        Privs:
            is_admin
            has_rootness
        DIT:
            CGI    dt sJob        { }
            NetscapeAdmServ    dt sJob      sABRT      { }
            Webmaster    sJob sHUP sABRT        { }
            www    dt sJob sHUP sABRT        { }
        Subtype: File 'file'
            { Default file type. }
            DDT:
                Webmaster e w r d n c    { }
        Subtype: Executable 'exec'
        { }
        DDT:
                Webmaster    e      { }
        Subtype: Directory 'diry'
            { }
            DDT:
                Webmaster    w r d n c      { }
Domain: www 'www#'
        {The www domain is where any Web servers
        will be run, such as the Commerce Server. }
        Privs:
        DIT:
            CGI    dt         { }
            www sUser sJob sHUP sABRT      { Needs
            to control itself. }
        Subtype: Config 'conf'
            { The httpd configuration file(s). }
            DDT:
                NetscapeAdmServ    w r    { }
                Webmaster    r w n c d      { }
                www r      { }
        Subtype: File 'file'
            { Default file type. This is the type created
            whenever the Commerce
            Server creates a file (log files, temporary files, etc.). }
            DDT:
                NetscapeAdmServ    r n    { }
                Webmaster    w r d n c      { }
                www    w r d n c      { }
        Subtype: Page 'page'
            { web pages }
            DDT:
                NetscapeAdmServ    r      { }
                Webmaster    r w n c d      { }
                www    r      { }
        Subtype: Directory 'diry'
```

-continued

```
        { }
        DDT:
            CGI    r      { }
            NetscapeAdmServ    w r    { }
            Webmaster    w r d n c      { }
            www w r d c { Web stores cached files in dirs,
                          needs to make and destroy as needed}
Domain: wwwCommon 'wwwc'
        { The common Web server domain
        (for items shared across burbs). }
        Privs:
        DIT:
        Subtype: Executable 'exec'
            { This is the Web Server executable type. }
            DDT:
                Webmaster    e      { }
                www    e      { Need to be able to run the server. }
        Subtype: Config 'conf'
            { }
            DDT:
                NetscapeAdmServ    w r d n c      { }
                Webmaster    w r d n c      { }
                www    r      { }
        Subtype: Directory 'diry'
            { }
            DDT:
                CGI    r      { }
                NetscapeAdmServ    w r    { }
                Webmaster    w r d n c      { }
                www    r      { }
        Subtype: Page 'page'
            { }
            DDT:
                CGI    r      { }
                NetscapeAdmServ    r      { }
                Webmaster    w r d n c      { }
                www    r      { }
```

[0483] Although the present invention has been described with reference to the preferred embodiments, those skilled in the art will recognize that changes may be made in form and detail without departing from the spirit and scope of the invention.

What is claimed is:

1. A secure commerce server system, comprising:

a plurality of regions or burbs, including an internal burb and an external burb, wherein processes bound to one burb cannot communicate directly to processes and data objects bound to other burbs, and wherein the internal burb includes a first protocol stack and the external burb includes a second protocol stack separate from the first protocol stack;

a commerce server, wherein processes and data objects associated with the commerce server are bound to the external burb;

an administration server, wherein processes and data objects associated with the administration server are bound to the internal burb; and

an assured pipeline connected between the external and internal burbs, wherein the assured pipeline controls transfer of information between the internal and external burbs such that a message from the commerce server to the administration server is routed up one or more layers of the second protocol stack and through the assured pipeline to the first protocol stack before being routed to the administration server.

2. The commerce server system according to claim 1 wherein the administration server includes means for reading and writing a configuration file used to configure the commerce server.

3. The commerce server system according to claim 1 wherein the administration server includes means for starting and stopping the commerce server.

4. The commerce server system according to claim 1 wherein the commerce server system further includes a CGI processor for executing CGI scripts, wherein the plurality of burbs further includes a cgix burb and wherein the CGI processor is bound to the cgix burb and all CGI scripts are restricted to operating within said cgix burb.

5. The commerce server system according to claim 1 wherein the assured pipeline includes means for enforcing a type enforcement mechanism to restrict communication between processes bound to different burbs.

6. The commerce server system according to claim 1 wherein the assured pipeline includes a proxy operating between burbs, wherein messages between burbs must pass through the proxy.

7. The commerce server system according to claim 6 wherein the assured pipeline further includes means for enforcing a type enforcement mechanism to restrict communication between processes bound to different burbs.

8. A method of conducting electronic commerce over a plurality of networks, including an external network and an internal network, wherein the internal network includes an administration server, the method comprising:

connecting a network interface to each of the plurality of networks, wherein the step of connecting includes the steps of connecting an external network interface to the external network and an internal network interface to the internal network;

defining a plurality of burbs, wherein the plurality of burbs includes a first and a second burb, wherein the first burb includes a first protocol stack and the second burb includes a second protocol stack separate from the first protocol stack;

assigning the external network interface to the first burb;

assigning the internal network interface to the second burb;

binding processes to burbs;

establishing an assured pipeline between the first and second protocol stacks;

receiving an electronic commerce request from the external network;

routing the electronic commerce request up one or more layers of the first protocol stack and through the assured pipeline to the second protocol stack; and

routing the electronic commerce request through the second protocol stack to the administration server.

9. The method according to claim 8, wherein binding includes preventing a process from accessing burbs other than the burb to which the process is bound and limiting

transfer of incoming packets such that incoming packets can only go to processes bound to the burb associated with the network interface the packet arrived on; and

wherein establishing an assured pipeline includes defining a proxy and ensuring that data passing between processes bound to the first and second burbs must pass through the proxy before moving between the first and second burbs.

10. The method according to claim 8, wherein each burb has its own routing table and wherein routing the electronic commerce request up one or more layers of the first protocol stack and through the assured pipeline to the second protocol stack includes examining an incoming packet to determine if its destination is an address in the first burb's routing table.

11. The method according to claim 8, wherein the step of binding processes to burbs includes the steps of:

defining a plurality of types;

assigning each data object to one of the plurality of types, wherein the step of assigning data objects includes defining type enforcement rules for accesses by processes to said data objects; and

applying a type enforcement check of accesses by processes to data objects.

* * * * *