CMSC 421
Final Project Design

David
12/10/17

1. Introduction
   1.1.System Description

The kernel modification I created adds several system calls that allow the root user to block individual tcp or udp ports, in either an incoming direction or outgoing direction. It also allows for root users to block individual files from being opened. If anyone else tries to call these kernel functions, they would receive a permissions error.

   1.2.Kernel Modifications

Kernel Files Modified:

- Linux/net/socket.c:
  - Syscall_define(bind, int, struct sockaddr__user*, umyaddr, int, addrlen)- Added an additional check function that detects if the socket trying to be bound is either tcp or udp. This blocks both incoming TCP and UDP requests.
  - Syscall_define(connect, int, struct sockaddr _user*, umyaddr, int, addrlen)- Added the same check as the one in bind. This one checks outgoing tcp requests.
  - Syscall_define(sendTo, …)- Uses the same check to block outgoing UDP requests.
- Linux/fs/namei.c:
  - int inode_permission(struct inode *inode, int mask)- Added a test to make sure that the inode that's permission is being tested isn't blocked. If it is, then the file cannot be opened.

Kernel Files Added:

- Linux/firewalls/port_block.h and Linux/firewalls/port_block.c:
  - void down_port_read(void)- Locks the list of blocked ports for reading. This allows any number of readers at once.
  - void up_port_read(void)- Unlocks the list of blocked ports for reading. This should always be called after down_port_read();
  - void down_port_write(void)- Locks the list of blocked ports for writing. Only one writer can work on the list at a time.
  - void up_port_write(void)- Unlocks the list of blocked ports after writing.
  - int check_invalid_port(int proto, int direction, unsigned short port)- Does basic error validation for system calls requiring root privileges.
  - int is_port_blocked(int proto, int direction, struct sockaddr_storage *umyaddr)- This is the function added to the system calls in socket.c. It checks whether the socket address is in the linked list, and returns EINVAL If it is.
  - asmlinkage long fw421_reset(void)- This deletes everything in the list of blocked ports, "resetting" the port blocking. It's only callable by the root.
  - asmlinkage long fw421_block_port(int proto, int dir, unsigned short port)- This adds the given port to the list of blocked ports, or an error if the port is already in the list. This is only callable by the root.

    - asmlinkage long fw421_unblock_port(int proto, int dir, unsigned short port)- This removes the given port from the list if it's already in it, or returns an error if it's not blocked. This is only callable by the root.
    - asmlinkage long fw421_query(int proto, int dir, unsigned short port)- This looks for the given port in the list of blocked ports, and returns how many times it was accessed if it's in the list. If it's not, it returns an error. This is only callable by the root.
- Linux/firewalls/file_block.h and Linux/firewalls/file_block.c:
    - void down_file_read(void)- Locks the list of blocked files for reading. This allows any number of readers at once.
    - /* Locks blocked_file linked list for writing */
    - void up_file_read(void)- Unlocks the list of blocked files for reading. This should always be called after down_file_read();
    - void down_file_write(void)- Locks the list of blocked files for writing. Only one writer can work on the list at a time.
    - void up_file_write(void)- Locks the list of blocked ports for writing. Only one writer can work on the list at a time.
    - int path_to_inode(const char* user path_name, ino_t* inode)- Converts the given path string into its corresponding inode value. This is only callable by the root user. This is a helper function for all the system calls in this file.
    - int check_invalid_inode(const struct inode *inode)- This checks to see if the inode is blocked. If it is, it returns -EINVAL. Otherwise, it returns 0. This is used in namei.c to check each file that's having open() called on it.
    - asmlinkage long fc421_reset(void)- This deletes every inode in the list of blocked files, "resetting" the file blocking mechanism. It's only callable by the root.
    - asmlinkage long fc421_block_file(const char *filename)- This function gets the inode of the file or directory passed into it, and stores it in the blocked files list. It returns an error if the file's inode value is already in the list. This is only callable by the root user.
    - asmlinkage long fc421_unblock_file(const char *filename)- This function removes the given file from the blocked files list, if the file is in it. Otherwise, it returns an error. It's only callable by the root.
    - asmlinkage long fc421_query(const char *filename)- This system call returns the number of times the given file has been accessed by processes on the computer, or an error if the file isn't blocked. It's only callable by the root user.

2. Design Considerations

2.1.Network Firewall

The network firewall essentially works by putting each blocked port into a linked list, and looking through that list each time a TCP or UDP request attempts to go through the bind(), connect(), or sendto() system calls. If the request is trying to bind to a port in the linked list, then the request would get blocked. Each blocked port number in the list is

paired with the direction it's supposed to be blocked in, and the protocol to be blocked on that port, as well as an access count value, which gets incremented each time a request gets blocked on that port. When the root calls fw421_block_port(), it creates a new entry into this linked list, and when they call fw421_ubblock_port(), it deletes the port from the list. I did it this way because the kernel's linked list API is really well documented and robust, and because I was told that a runtime of O(n) was good enough.

The locking mechanism I used to keep everything thread-safe is fairly simple. I used two mutexes, in conjunction with an integer, to allow for either any number of readers to have access to the linked list, or exactly one writer, but not both. Whenever a reader wants to check out the linked list, it calls up_port_read(), which first locks the reader mutex. It then increments the reader count, and checks to see if the reader count is one. If it is, then the reader must lock the writer mutex, to keep writers from being able to access the linked list while readers are using it. It then unlocks the reader mutex. When the reader is done, it locks the reader mutex, decrements the reader count, unlocks the writer mutex if the reader count is zero, and unlocks the reader mutex. The writer mutex is used like a normal mutex. I did it this way because it was the simplest reader-writer locking scheme I found, and it works pretty well.

2.2.File Access Control

The setup for the file access control is nearly identical to the network firewall. All the functions for this part of the project are prototyped in linux/firewalls/file_block.h and implemented in linux/firewalls/file_block.c. When a file is blocked using fc421_block_file(), it calls path_to_inode() on the path name passed into it. This function finds the inode id of the file, which is different for each one. fc421_block_file() then adds that inode number into the linked list that holds all the blocked file inode numbers, which is used to block them from being accessed in the inode_permission() function in Linux/fs/namei.c. This works great for files that already exist, even making the file explorer show a lock and large X over blocked files, but does not work on files that don't exist yet. Unfortunately, I never got that part of the project working. I did it this way because I noticed that the most far-reaching permissions function that I could find was inode_permission(), but I only noticed too late that only existing files have traceable inode numbers that I could store in the linked list.

The locking mechanism is exactly the same as the network firewall, and the system calls all work the same as the network firewall, for the most part. It uses a reader-writer locking system that's exactly the same as the network firewall, and the linked list system works the same, but with different structures within them. This was intentional, since it's easy to both write and understand similar systems.

3. System Design

3.1.Network Firewall

The network firewall has four static variables that are defined on startup: a reader and writer mutex, a volatile int for keeping track of the reader count, and a linked list LIST_HEAD. The locking setup was explained earlier in section 2.1, but I'll go over it again here. The reader lock is done using the reader count int, which is guarded by the

reader mutex. When the reader count goes from zero to one, it locks the writer mutex until the reader count goes back to zero. The writer mutex works exactly as a normal mutex would: when a write is about to happen, it locks the writer mutex, and then unlocks the mutex when it's done.

The project implements the linux kernel linked list structure, which works by having a list_head struct in each node of the list, and uses preprocessor functions to add or loop through it. The basic node structure contained in the linked list holds the prototype to be blocked on the node (either IPPROTO_TCP for tcp requests or IPPROTO_UDP for udp), the direction in which the port should be blocked (0 for outgoing requests, 1 for incoming), the actual port number, the access count of the blocked protocol on the port, and a list_head struct to make it able to be a part of the linked list. When a new port is blocked, it gets added to the linked list, and when it gets unblocked, it gets removed from the list.

The actual checking within the system calls are also fairly simple. Within each of the system calls I modified, there's a section in which the function looks up the socket associated with the fd int given by the user. After this successfully completes, I inserted a call to is_port_blocked(int proto, int direction, struct sockaddr_storage *umyaddr), using the socket's protocol type, the correct direction for what the system call is doing (1 for bind(), 0 for connect() and sendto()), and the address struct. If the return value of this function isn't 0, then the system call doesn't go through, and returns an error. The first thing the function does is to convert the protocol enum stored in the socket into its appropriate ip protocol. If the protocol isn't UDP or TCP, then the function returns 0, since it shouldn't be blocked then. The next thing is_port_blocked does is to extract the port trying to be bound to, which is located in the address struct. Normally, this isn't a native member variable of a sockaddr_storage struct, but it can safely be cast to a sockaddr_in, which does has a sin_port member variable. The port number is stored in network endianness though, which can be different from the native computer endianness, so that has to be converted using one of the endian macros defined in the kernel. Once the protocol, direction, and port numbers are all found, the function locks the linked list for reading, and searches through the list of blocked ports to find any matches. If there is one, it unlocks the linked list from reading, locks the linked list for writing, adds one to the access count of that node, and returns –EINVAL. Otherwise, if it looks through the whole thing and doesn't find any matches, it returns 0, signifying that everything's fine.

Fw421_reset(), like every system call in port_block.c, first checks if the root is the one calling the function, by comparing the current_id().val to zero. If it's not zero, then it returns –EPERM, since the caller wasn't the root. Next, it locks the linked list for writing, then loops through the whole list, deleting each node as it goes. At the end of the function, it unlocks the linked list, and returns 0.

Fw421_block_port() starts by checking to see if the user is root, and checking to see if the given parameters for the port to be blocked aren't invalid. If everything's good, it then locks the linked list for reading, and makes sure the port with the given parameters isn't already blocked. If it wasn't, it unlocks the list from reading, kmallocs a new blocked_port struct, and fills in the structs' member variables with the parameters passed into the function. It then locks the list for writing, and adds the new struct to it. Finally, it unlocks the list and returns 0.

Fw421_unblock_port() does the same error checking done by fw421_block_port() to start with. It then locks the list for reading, and loops through the linked list until it finds

something that matches the given parameters. Once it does, it unlocks the linked list from reading, and locks it for writing. Then, it deletes the matching file from the list, unlocks the list from writing, and returns 0. On a failure to find a matching port, it returns –ENOENT.

Fw421_query() works almost exactly the same as Fw421_unblock_port(), except if doesn't delete the node it finds. Instead, it returns its' access count. Due to the fact that this doesn't change anything in the linked list, it also never locks for writing either.

3.2.File Access Control

The static variables defined in startup for the file access control in Linux/firewalls/file_block.h are exactly the same as the ones found in the network firewall, and the locking mechanism is the same as well. The main difference between the two is in how the blocking is checked within the kernel. Instead of modifying the functions that do the file opening themselves, since there's so many of them, I modified inode_permission() in fc/inode.c, which checks to see if the inode passed into it is read-only, or if the file is immutable in some way. Every file opening function in the kernel calls this at least once before opening the file, so it was a great way to block existing files. Unfortunately, this function doesn't take a path string, it takes an inode, and those are made when a file is created for the first time, so it's impossible to preemptively block a file using this method, so this project can't do that. The function check_invalid_inode() was added to inode_permission(). This function loops through the blocked_file linked list, and returns an error if any of the inode values stored in the list match the inode value of the file trying to be opened.

Whenever a function in file_block.c takes in a filename string, the first thing it does is call path_to_inode(const char* __user path_name, ino_t* inode) on it, which does root user checking and parameter checking, then uses kern_path() to get the path struct of the given file string. From the path struct, we can get the file's inode by going through the path's dentry, which contains an inode struct. This inode struct has the inode id it represents stored as its inode value, which is what path_to_inode() returns. If the file lookup failed, it returns an appropriate error message.

The inode value is what's stored within the blocked_file linked list. Each blocked_file struct holds the inode number, the access count of the blocked file, and a list_head struct, so it can be a part of the linked list. The inode value stored is the one found by using path_to_inode(), as shown above.

All the system calls is this section are written almost identically to their counterparts in the network firewall, except for the fact that they call path_to_inode() at the beginning, and only have to compare inode values while looping through the blocked_file linked list.

4. References

Roman10, A. (n.d.). Linux Kernel Programming–Linked List. Retrieved December 10, 2017, from http://www.roman10.net/2011/07/28/linux-kernel-programminglinked-list/

Stack Overflow - Where Developers Learn, Share, & Build Careers. (n.d.). Retrieved December 10, 2017, from https://stackoverflow.com/

B. (n.d.). Beej's Guide to Network Programming. Retrieved December 10, 2017, from http://beej.us/guide/bgnet/