CMSC 421
Final Project Design

Nimit Patel
12 May 2019

1. Introduction

1.1.System Description

We had to create a network firewall and a file access control mechanism. The root had to be able to add ports on the basis of direction and protocols in the data structure. If the port was in the firewall, then the access had to be blocked. Similar thing had to be done with the file access control. The root had to be able to add file names and the access to those files had to be blocked.

1.2.Kernel Modifications

List the files (and functions within files) that you modified as part of your implementation. Be sure to include the list of files that you added to the kernel as part of this list as well. For modified files, include a very brief description (single sentence per function modified) of what was modified in the file. You will be describing your changes more in-depth in a future section of this document. This section should essentially be formatted as a list.

- fs/namei.c - *path_init* function was modified; char* s is passed into the function in fcManager.c to check if it's valid

- net/socket.c - *__sys_connect* and *__sys_bind* functions were modified; port, protocol, and direction were extracted and sent to the function in fwManager.c to check if it's valid

2. Design Considerations

2.1.Network Firewall

I used the built-in red-black tree to store the data. Linked list would not have been efficient because if the user added port 65000, then it would need to iterate 65000 times. At first, I was focused on building the data structure. After the data structure was tested, I modified the kernel.

I added two system calls, *fw421_check_blocked(proto, dir, port)* and *fw421_print_ports(). Check_blocked* function calls the function in the data structure that is called from within the kernel. It would increment the counter if the access is not allowed. *print_ports()* uses printk to print all the content of the data structure. The driver for the new system calls are also created.

The code is in the folder called *proj2fw*. *fwSyscalls.c* contains the system calls. *fwRBTree.c* contains the data structure. *fwManager.c* contains the function that is called from the kernel code.

2.2.File Access Control

I used the built-in hash table to store the data. If there is collision, then the item is added to a linked list that is at that location. The hash table has 65423 spots, so there could be a linked list attached to each of those spots.

Same as the firewall, I added two new system calls: *fc421_check_blocked(filename)* and *fc421_print_files()*. *Check_blocked* calls the

function that would be called by the kernel code. *Print_files* uses printk to print out all the content of the data structure.

The code is in the folder called *proj2fc*. *fcSyscalls.c* contains the system calls. *fcHT.c* contains the data structure. *fcManager.c* contains the function that is called from the kernel code.

3. System Design

3.1.Network Firewall

The struct *BlkPort* is used to store the data related to the port that is being blocked. The struct contains *index*, *attempts*, and *node*. *Index* contains the *port*, *protocol*, and the *direction*. I needed a way to differentiate one *BlkPort* from another so that I could place them on the tree. Port is 16 bits. Protocol and direction are 1 bit each. Total is 18 bits. This could be stored in one int (32 bits). The int would look like:

XXXXXXXX XXXXXXPP PPPPPPPP PPPPPPRD

P is the 16 bits used to store the port. R is the bit used to store the protocol. D is the bit used to store the direction. This creates a unique integer for all combination of port, protocol, and direction. This makes it easier to compare two *nodes*. *getIndex()* is used to create an *index*. *getDirection()* and *getProtocol()* returns an direction and protocol from the *index* that was passed in the arguments. They perform simple bit-wise operations.

Reference (2) and (17) were helpful in creating the red-black tree. Some modification was needed to implement the functions for insert, find, and remove. The functions for *query*, *reset*, *check_blocked*, and *printPorts* had to be written from scratch. *Query* uses the *find* function to get the struct and returns the *attempts* element from the struct. *Check_blocked* is like *query* but it increments the *attempts* counter if found. *Reset* has a while loop that goes through all the *nodes* and deallocates them after disconnecting them from the tree. *PrintPorts* is similar to reset except it prints the content.

The system calls are in *fwSyscalls.c* file. They first check if the user is root. Then, they convert the protocol to either 1 or 0. If the user is using IPPROTO_TCP, then the integer is 6. If the user is using IPPROTO_UDP, then the integer is 17. To fit this into the data structure, it needs to be either 1 or 0. 1 is used to represent TCP and 0 to represent UDP. After doing that, the system call gets the index and passes that index to their respective functions. Finally, the return value from those functions is passed back to the user.

fwManager.c contains the function that is called by the root. Initially, I thought if the data had to be modified before being sent to the *check_blocked* function of the data structure, then it would be better to place it in a separate file. When I passed in *sockaddr* into the function, it gave me an error saying incomplete struct so then I just extracted port from sockaddr and passed it in.

I modified the *__sys_connect* function (~line 1680). *Connect* is used to connect to a device over the internet so it's outbound. Thus, the direction is 0. Then I got the protocol from *sock->type*. If it's SOCK_DGRAM, then it's UDP and TCP if it's SOCK_STREAM. Port was obtained by casting *sockaddr_storage* to *sock_addr_in* and getting the element *sin_port*. Since it's in different byte order, I used *ntohs()* to convert it into the system byte order.

__sys_bind function (~line 1490) was also modified. It was very similar to connect except direction was set to 1.

3.2.File Access Control

I implemented a hash table to store the data because implementing a tree with a string would have been more complicated. I used the SDBM hash function from (14) because it said that it would give good distribution of indices. The struct *BlkFile* contains *filename*, *count*, *key*, and *node*. *Count* is the number of times the file was attempted. *Key* is the hash. *Node* is used by the kernel hash table.

It was much more difficult to implement the hash table than the red black tree because there weren't many good sources. (7) and (8) helped me implement the hash table. In this hash table, there is a linked list at each entry in the hash table so if there is collision, then the node is added to the linked list.

Insert is simple with *hash_add* function. *Remove* is similar but with *hash_del* function. Then there are two more function: *hash_for_each* and *hash_for_each_possible*. *hash_for_each* returns everything in the hash table. *hash_for_each_possible* returns everything at that particular index. If there was a collision at that index, then *hash_for_each_possible* would return more than one node. *hash_for_each_possible* is used by *find*. *Find* is then used by *insertFile*, *removeFile*, *queryFile*, and *checkFileBlocked* functions. *hash_for_each* is used by *resetHT* and *printFiles*.

In *reset* function, *hash_for_each* is like a for loop so when I had to delete, I deleted element at the previous index. Because if I had deleted the current file, then it wouldn't have moved on to the next file so the table wouldn't really be empty. I deleted file[i - 1] and the i^{th} iteration.

In *fwSyscalls.c*, I use the *copyFile* function to copy the userspace filename to the kernel space. It also checks whether the name exceeds the allowed Linux file length(4096).

In the kernel, I modified the *path_init* function (~line 2160). The string s contains the *filename*, so it's passed to my function in *fcManager.c* to before being returned. If 'dummyDir' is blocked, then it will block access. It doesn't work as well for absolute path. The struct *nameidata* holds the *path*. I would have passed the path into function *d_path* to get the path in string but it wasn't working so I just used the *filename* instead. *link_path_walk* is a name resolution function that converts path name into *dentry*.

4. References

1. (n.d.). Retrieved from http://www.cse.yorku.ca/~oz/hash.html

2. (n.d.). Retrieved from https://www.kernel.org/doc/Documentation/rbtree.txt

3. (n.d.). Retrieved from http://beej.us/guide/bgnet/

4. (n.d.). Retrieved from https://www.kernel.org/doc/htmldocs/filesystems/API-d-path.html

5. (n.d.). Retrieved from https://www.win.tue.nl/~aeb/linux/lk/lk-8.html

6. (n.d.). Retrieved from http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html

7. (n.d.). Retrieved from https://kernelnewbies.org/FAQ/Hashtables

8. A generic hash table. (n.d.). Retrieved from https://lwn.net/Articles/510202/

9. Cnicutarcnicutar 148k19280330. (n.d.). Hash function for string. Retrieved from https://stackoverflow.com/questions/7666509/hash-function-for-string

10. Forhappy. (n.d.). Forhappy/rbtree. Retrieved from https://github.com/forhappy/rbtree/blob/master/rbtree.c

11. Getting absolute path of a file. (n.d.). Retrieved from https://www.linuxquestions.org/questions/linux-kernel-70/getting-absolute-path-of-a-file-823726/

12. Jamesroutley. (2017, August 24). Jamesroutley/write-a-hash-table. Retrieved from https://github.com/jamesroutley/write-a-hash-table/tree/master/02-hash-table

13. Linux source code: Arch/x86/entry/syscalls/syscall_64.tbl (v4.18.6). (n.d.). Retrieved from https://elixir.bootlin.com/linux/v4.18.6/source/arch/x86/entry/syscalls/syscall_64.tbl

14. Partow, A. (n.d.). General Purpose Hash Function Algorithms - By Arash Partow. Retrieved from http://www.partow.net/programming/hashfunctions/

15. Pathname lookup in Linux. (n.d.). Retrieved from https://lwn.net/Articles/649115/
16. SiddhantSiddhant 1, & Cafcaf 194k27255395. (n.d.). How can I get a filename from a file descriptor inside a kernel module? Retrieved from https://stackoverflow.com/questions/8250078/how-can-i-get-a-filename-from-a-file-descriptor-inside-a-kernel-module
17. Trees II: Red-black trees. (n.d.). Retrieved from https://lwn.net/Articles/184495/