CMSC 421
Final Project Design

Matthew Manzi
12 May 2019

1. Introduction
   1.1. System Description

The task of this project was to create two security systems: a firewall and a file access control system. The firewall system is a Layer 4 (of the general OSI model) firewall, that is to say it blocks communication involving the TCP and UDP Layer 4 protocols. The file access control system blocks files in the filesystem by interpreting path names. Each system has its own internal structure with which it keeps track of currently blocked entities. These systems are entirely original additions to the kernel—complete with eight new system calls—and do not build off of functionality of SELinux, programs like `iptables`, or the like. Both systems are designed for role of system administrators, specifically ones that have permission to act as the `root` user.

1.2. Kernel Modifications

- arch/x86/entry/syscalls/syscall_64.tbl
  - Eight entries were added to the system calls table, one for each of the system calls in fw/fw_421.c.
- fs/namei.c
  - `link_path_walk()`
    - Here, a check was added right before this function returns a success to determine if a resolved path name has been blocked, in which case the function then returns a failure.
- fw/fw_421.c
  - `port_insert()`
  - `port_search()`
  - `file_insert()`
  - `file_search()`
  - `erase_ports_tree()`
  - `erase_files_tree()`
  - `fw421_reset` system call
  - `fw421_block_port` system call
  - `fw421_unblock_port` system call
  - `fw421_query` system call
  - `fc421_reset` system call
  - `fc421_block_file` system call
  - `fc421_unblock_file` system call
  - `fc421_query` system call
- fw/Makefile
- include/fw/fw_421.h
- include/linux/syscalls.h
  - Eight function prototypes, corresponding to the system calls in the syscall_64.tbl file, were added to the bottom of this file, just above the last `#endif`.

- Makefile
    - The name of the new system calls' directory was added to the `core-y` Makefile variable.
- net/socket.c
    - `__sys_bind()`
        - Here, a check was added right after the function copies the required information to kernel space, where the protocol and port number are determined and combined with an "incoming" flag to then check for (and subsequently block) a matching port entry.
    - `__sys_connect()`
        - Here, a check was added right after the function copies the required information to kernel space, where the protocol and port number are determined and combined with an "outgoing" flag to then check for (and subsequently block) a matching port entry.
    - `__sys_sendto()`
        - Here, like in `__sys_connect()`, a check was added right after the function copies the required information to kernel space, where the protocol and port number are determined and combined with an "outgoing" flag to then check for (and subsequently block) a matching port entry.
    - `___sys_sendmsg()`
        - Here, like in `__sys_connect()`, a check was added right after the function copies the required information to kernel space, where the protocol and port number are determined and combined with an "outgoing" flag to then check for (and subsequently block) a matching port entry.

2. Design Considerations

2.1. Network Firewall

The firewall is made up of a tree-type data structure to store data, four system calls to read and write data, and the four separate checks added to net/socket.c (mentioned above). It fully supports blocking and unblocking individual port entries, clearing all blocks, and querying the number of attempts made on each blocked port entry. Each port entry is defined by the three-tuple of the protocol (TCP or UDP), the direction (incoming or outgoing), and the port number being blocked.

To store firewall entries, I elected to use the Linux kernel's red-black tree implementation, because it is well-documented, port numbers are easily sortable, and its worst-case scenarios are always O(log(n)) for insertion, search, and deletion. Still, the firewall introduces an interesting problem. Because port blocks are categorized by the aforementioned three-tuple, all three must be used as criteria for storage and lookup. I achieved this by storing only one red-black tree entry per port and setting one or more of four flags in the entry to indicate which three-tuple(s) is blocked. Necessarily, each entry

stores four attempt counts, one for each possible three-tuple for that port number. To access this structure from kernel checks, the red-black tree and its relevant search function are exposed to the rest of the kernel.

To ensure thread safety, I used the kernel's reader-writer lock system. It greatly simplifies thread safety. I implemented the locks for reader priority, since only the query system call is a reader, thus there are likely to be many fewer readers that could starve out writers.

2.2.File Access Control

The file access control system is also made up of a tree-type data structure to store data and four system calls to read and write data, but it only requires a single check added to fs/namei.c (mentioned above). It, similarly, supports blocking and unblocking individual file (or directory) entries, clearing all blocks, and querying the number of attempts made on each blocked file entry. Each file entry is defined by the absolute path to the file and keeps an associate number of access attempts stored with the path name.

Directories can be blocked just as files can. Convention requires that directories may be referenced with or without a trailing forward slash character (/), indicating that the file is indeed a directory. However, no matter how the directory is blocked and no matter how it accessed, the trailing slash will not affect the access control system. When blocking a directory, files and subdirectories within the blocked one will not be blocked. If the opposite were true, the system would have to start making unwarranted assumptions about what the user desires. One shortcoming of the access control system is that symbol links. I recommend that system administrators disallow users from making symbolic links until a later release.

To store file entries, I elected to use the kernel's red-black tree implementation here, as well, for the same reasons as with the firewall, with the added motivation that red-black trees scale very well. To access this structure from kernel checks, the red-black tree and its relevant search function are exposed to the rest of the kernel.

To ensure thread safety, I once again used the kernel's reader-writer lock system. I implemented these locks for reader priority, too.

3. System Design

Shared between both systems is a header file (include/fw/fw_421.h) which declares each one's red-black tree, defines the custom structs stored in those red-black trees, and declares the custom insert and search functions for those red-black trees. The header file also defines macros for the firewall system.

3.1.Network Firewall

A `ports` red-black tree is defined in fw/fw_421.c as an empty red-black tree, using the existing macro. It is the designed to hold the custom `blocked_port` struct, which has three data members, aside from node information. It holds an unsigned short integer for the port number, a one-byte unsigned character for the flags, and array of four long integers for the count of attempts made on that port, depending on the three-tuple that was blocked and accessed.

There are four valid flags, defined as macros in the header file, for a port entry which are the values (in base-10): 1 for outgoing TCP, 2 for incoming TCP, 16 for outgoing UDP, and 32 for incoming UDP. These correspond to individual bits in the stored flag, allowing bitwise operations to set, compare, and remove them from the entry's member. These flags are also used to access the respective attempt counts in the entry's array member. First, the user must check if the flag is one of the UDP flags in the higher nibble of the byte. If so, they will bitwise shift to the right by 4 bits and add 1 to the result: this is the index. If not, simply subtract 1 from the flag value and access the attempt of one of the TCP entries.

Resetting the firewall, simply checks if the tree is not yet empty and calls the `erase_ports_tree()` helper function. This function iterates the tree's nodes in order and will erase and free them one by one. If the caller was not `root`, -EPERM is returned.

Blocking a port is simply adding it to the red-black tree. The calling user must be `root`, otherwise -EPERM is returned. The protocol must be one of IPPROTO_TCP or IPPROTO_UDP, otherwise -EINVAL is returned. The direction must be one of 0 (for outgoing) or 1 (for incoming), otherwise -EINVAL is returned. The port number must be in the inclusive range [1, 65535], otherwise -EINVAL is returned. All of these will cause the block to be aborted and the error will be logged to the kernel log. Once the protocol is determined, it is assumed to be outgoing and the flag is chosen accordingly. After the direction is verified, the flag is shifted to the left by the number of bits corresponding to the direction parameter. A new struct is allocated with the determined parameters and passed into `port_insert()` to either be placed into the tree or to update the flags on an existing node with the same port number. If that function finds a duplicate entry, -EEXIST will be returned.

Unblocking a port has the same process as blocking does until it determines the flag and uses the `port_search()` function to get a pointer to the tree entry and do one of three things: 1) it will receive a NULL pointer and return -ENOENT since the port with the requested three-tuple is not blocked; 2) it will see that the only flag set on the entry is the one that was requested and so it will delete the entire entry; 3) it will see that the requested flag is not the only set on the entry, remove the flag from the entry, and set the respective attempts count to zero.

Querying a port is very similar in the parameter checks that it performs. It also performs the search like unblocking does and returns the same -ENOENT when the port is not found. Otherwise, it simply returns the attempts count based on the determined flag (up to LONG_MAX).

In four function in net/socket.c, four kernel checks were added, one per function, like mentioned previously. The checks correspond to bind, connect, sendto, and sendmsg function and system calls. The kernel check code in each of them is identical, each placed after the kernel copies the address struct to kernel space. The only difference is in sendto, where the check must also be inside the `if (addr)` statement to ensure it is only blocking requests from the sendto system call, and not also the send system call. First, check if the tree is empty, no reason to waste any more compute time if it is. Next, get the protocol from the socket and the port number from the `sockaddr` struct, using the proper casting based on the struct's sa_family member's value. The direction is implied from the enclosing function: bind is incoming and connect, sendto, and sendmsg are all outgoing. Once the three-tuple has been constructed into port and flag pair, they are passed into `port_search()`. The result, if present, will have its respective attempt count incremented by 1 and the enclosing functions will return -EPERM.

3.2.File Access Control

A `files` red-black tree is defined in fw/fw_421.c as an empty red-black tree, using the existing macro. It is the designed to hold the custom `blocked_file` struct, which has two data members, aside from node information. It holds a pointer to a C-string for the absolute path name and a long integer for the count of attempts made on that path name.

Resetting the file access control system, simply checks if the tree is not yet empty and calls the `erase_files_tree()` helper function. This function iterates the tree's nodes in order and will erase and free them one by one. If the caller was not `root`, -EPERM is returned.

Blocking a file is simply adding it to the red-black tree. The calling user must be `root`, otherwise -EPERM is returned. The pointer to the path name must not be NULL, otherwise -EFAULT is returned. The path name must start with slash character (/), otherwise -EINVAL is returned. All of these will cause the block to be aborted and the error will be logged to the kernel log. The path name is copied then from user-space. A new struct is allocated with the path name, stripped of a trailing slash, and passed into `file_insert()` to be placed into the tree. If that function finds a duplicate entry, -EEXIST will be returned.

Unblocking a file has the same process as blocking does but instead of creating an entry, it uses the `file_search()` function to get a pointer to the tree entry and do one

of two things: 1) it will receive a NULL pointer and return -ENOENT since the path name is not blocked; 2) it will receive a valid pointer and delete the entry.

Querying a file is very similar in the parameter checks that it performs. It also performs the search like unblocking does and returns the same -ENOENT when the file is not found. Otherwise, it simply returns the attempts count based on the determined flag (up to LONG_MAX).

In `link_path_walk()` in fs/namei.c, one kernel check was added. The check is inside the `!nd->depth` if-statement, immediately preceding the `return 0` statement. First, check if the tree is empty, no reason to waste any more compute time if it is. Next, calloc a buffer for the path name to check, with the size of the kernel's PATH_MAX macro. Then use the `dentry_path_raw()` function to get the currently resolved path name's absolute path. This directory entry does not point to the last component of the path, so it a slash (`/`) and then the value of `nd->last.name` are concatenated onto the end of it to produce a full absolute path name. A trailing slash, if present, is removed and then searched for in the tree structure. If it is found, a -EPERM is returned from `link_path_walk()` and the attempts count is incremented by 1.

4. References

"Bitwise Operators in C/C++." *GeeksforGeeks*, 28 Mar. 2019, www.geeksforgeeks.org/bitwise-operators-in-c-cpp/.

Bolton, David. "What Does Unsigned Mean in Computer Programming?" *ThoughtCo*, ThoughtCo, 8 Jan. 2019, www.thoughtco.com/definition-of-unsigned-958174.

"Debugging by Printing." *Debugging by Printing - ELinux.org*, elinux.org/Debugging_by_printing.

"External Variables." *The C Programming Language*, by Brian W. Kerninghan and Dennis M. Ritchie, Prentice Hall, 1998.

"Pathname Lookup¶." *Pathname Lookup - The Linux Kernel Documentation*, www.kernel.org/doc/html/latest/filesystems/path-lookup.html.

"Printf." *Cplusplus.com*, www.cplusplus.com/reference/cstdio/printf/.

Summer_More_More_TeaSummer_More_More_Tea 8, and devnulldevnull 86.4k23166186. "How to Get Full Pathname from Struct Dentry in Linux Kernel." *Stack Overflow*, stackoverflow.com/questions/17216856/how-to-get-full-pathname-from-struct-dentry-in-linux-kernel.

Torvalds. “Torvalds/Linux.” *GitHub*, github.com/torvalds/linux/blob/master/Documentation/rbtree.txt.

Tutorialspoint.com. “C Constants and Literals.” *Www.tutorialspoint.com*, www.tutorialspoint.com/cprogramming/c_constants.htm.

Tuxthink. “Module to Display the Current Working Directory.” *Module to Display the Current Working Directory*, tuxthink.blogspot.com/2011/12/module-to-display-current-working.html?m=1.