Possible Vulnerabilities Accounted for in Assignment 2 Part 1

Most of the common vulnerabilities that I found in Assignment 2 Part 2 were accounted for in my reference monitor submitted in Assignment 1 Part 1. I will briefly mention these vulnerabilities and how they are accounted for in the latest version of my reference monitor.

The first vulnerability I considered was a write to a negative offset. In my reference monitor, if the writeat() call sees a call to write to a negative offset, then it calls the sandbox writeat() to a negative offset so that the sandbox exception for the invalid write will be thrown.

The second vulnerability I considered was whether an invalid write would clear the contents of the buffer. According to the specs for the reference monitor, if a valid write is called, and then an invalid write is called, then the first valid write should still be written to the file after a second valid write is called. My reference monitor handles this case by making writes to the files and the buffer only after all possible invalid cases have been checked.

The third vulnerability that I considered was the possibility of a writeat() to a closed file when the buffer is empty. My reference monitor handles this case by keeping a variable that tracks if the file has been closed and denying the writeat() call if the variable states that the file had previously been closed.

The fourth vulnerability that I considered was a multithreading vulnerability where an undo() call in one thread would delete the valid buffer being used by a writeat() call in another thread. This would raise an exception if the contents in the buffer are deleted (by undo()) right before the reference monitor writeat() calls the sandbox writeat() with the empty buffer contents as the arguments. My reference monitor handles this case by having all calls wait for a shared lock at the beginning of every call. The lock will always be released after a reference monitor call because the readat(), writeat(), and close() functionality is put in a try block and the shared lock release call is placed in a finally block.

Vulnerability Fixed in Assignment 2 Part 3

The only vulnerability that I had in Assignment 2 Part 1 was not tracking the EOF in my reference monitor. My reference monitor would allow the user to commit a write with an out-of-bounds offset into the buffer because my writeat() function did not check if the offset of a writeat() call is valid before placing the write in the buffer. For example, if a user initialized an empty file and performed the call writeat(“123”, 16), then my reference monitor would place this write into the buffer despite the EOF being at position 0.

To fix this vulnerability, I initialized two variables: self.eof and self.true_eof. The first variable, self.eof, tracks the current working EOF. This is the EOF that will be used to determine if a writeat() exceeds the EOF. This is equal to either the current EOF of the file or the EOF of the write in the buffer, whichever is greater. The second variable, self.true_eof, tracks the EOF that is actually in the file. This variable is needed so that we have a value to reset self.eof to if the user calls undo() on an uncommitted write. When the file object is initialized, the true EOF is obtained by getting the length of a readat(None, 0) call. The current working EOF is equal to the true EOF at this point because there is nothing in the write buffer. Then, both EOF variables are updated as writeat() and undo() calls are made.

In writeat(), if a call is being made that exceeds self.eof, then the function calls the sandbox writeat() to write an empty string to the location self.true_eof+256 to purposefully trigger the sandbox’s out-of-bounds offset write exception. This is done rather than raising the exception from the reference monitor itself to ensure that the reference monitor always matches the exception handling behavior of the sandbox file operation calls.

The eof variables are updated in two places in the reference monitor writeat() function. First, self.true_eof is updated to equal self.eof, the EOF of the committed write, whenever a valid sandbox writeat() call is used. The self.true_eof variable should only be updated whenever a sandbox writeat() is called because only the sandbox writeat() can alter the true EOF of the file that the file object manages. Second, self.eof is updated to either the EOF of the new uncommitted write or the current value of self.eof, whichever is bigger, after the new write is stored in the buffer.

In undo(), the self.eof variable is set to the value of self.true_eof. This means that if an uncommitted write had extended the working EOF of the file object, the EOF is reverted to its true location when the uncommitted write is reverted.