Jeffrey Wong
JW4186
Assignment 2 Part 3

In this assignment we were tasked to write a reference monitor that implements an undo() function within the file operations in Repy. The standard writeat() operation writes data into an opened file, but undo() adds the ability to effectively cancel the most recent writeat() call. When writeat() is called, the file should not immediately write the contents of that call into the file, rather it should wait for confirmation in the form of a subsequent writeat() call or a close() call to close the file. At that point it writes the contents of the original writeat() into the file and either continues the cycle or closes the file. We had already created an implementation of this reference monitor in part 1 of this assignment, but it became clear that there were many scenarios in which this functionality would be compromised when we all tried to write code to attack each other's reference monitors in part 2.

My original implementation of the reference monitor was a very simple one. In addition to storing the pending data and offset position for the unconfirmed and not yet written data, the only alterations I made were for writeat() to write the previously pending contents if there was any, and for close() to write any remaining pending data. I thought that it was best to keep things simple, since the fewer operations there are the fewer mechanisms there are for attackers to exploit. However, my original implementation failed to account for threaded scenarios where multiple operations are operating at the same time. Data might be compromised or altered in an unexpected way if there are multiple threads operating on the same objects at the same time. Many attacks from other students included multiple threads to attempt to trigger this kind of race condition. Therefore, a lock had to be added to the file class. In each file operation, after the validity of the operation itself has been established, that operation acquires the lock for the object it is acting on. When one thread is holding a lock for an object, no other threads can operate on that object or acquire that lock. This maintains the integrity of the object and allows the entirety of the operation to be completed without any unexpected actions from other threads that could deviate the outcome. After the operation is done using the object, the lock is released and another thread can now acquire the object if it has been waiting for it. Referencing the reference monitors from other students in the class, I added this lock acquisition action to readat(), writeat(), undo(), and close().

The implementation of the lock mechanism greatly reduced the amount of attacks that were able to successfully bypass my reference monitor, but there were still many errors that managed to pop up. One of them is that the reference monitor needed to track the state of the information on disk. My original implementation checked if there was any data in the buffer before performing a write operation, but it was more efficient and secure to add a boolean property to the file class itself denoting whether there was content in the buffer, so I added a buffer_flushed bool to the class and added code to writeat() and undo() to set its value appropriately. Another issue in many attacks was calling file operations after the file had already been closed. To account for that I added code in readat(), writeat(), and undo() to raise a FileClosedError if the file object itself did not exist. A big issue that arose in many attacks is when operations are called, the user can input invalid values. Some cases are simpler to take care of, such as negative values, where I added code to raise RepyArgumentErrors if the offsets in writeat() or readat() calls or the bytes in readat() calls were negative. I also added code to raise an RepyArgumentError if the data in a writeat() call was not a string. The more complicated scenario was when the user input an offset value greater than the size of the file. To account for this I had to track the size of the file by adding a filesize property to the file class, adding code to raise a SeekPastEndOfFileError if the offset in the call was larger than filesize, and updating filesize after each write operation. After making all of the above changes, I ended up with a much more complicated reference monitor which is more similar to what some other students implemented but was able to withstand all valid attacks when I tested it.