Upon analysing the attack matrix and examining the successful attacks that bypassed my reference monitor, I have come to the realisation that there are significant aspects concerning pointer arithmetic and memory tracking that I overlooked during the initial construction of my monitor. Initially, I failed to recognize the crucial role of the offset value, which determines the position where incoming input characters are stored in memory. Neglecting to ensure that the offset begins at position '0' each time the program is initiated led to a situation where new input was stored with a non-zero offset. This oversight presents a substantial security risk, as it allows attackers to craft input in a manner that enables them to execute and extract data from memory they shouldn't have access to. Such vulnerabilities can lead to dangerous buffer overflow attacks. I fixed this by introducing a check to ensure the file is written starting from the exact position it needs to.
Secondly, I neglected to monitor whether the file being written to was already closed. This oversight created a problem as there is no file the data is being written to. To address this issue, I implemented a solution by ensuring I tracked the file's status every time the write function was called. If the file was found to be closed, I introduced a mechanism to raise a File Close error, allowing me to detect and handle situations where the file was not accessible.
Furthermore, another critical flaw within my reference monitor, closely linked to the issue mentioned earlier, was my failure to monitor the offset, allowing it to go negative. Allowing the offset to turn negative creates an alarmingly hazardous scenario, wherein data stored in the memory, intended to be inaccessible to users, becomes exposed. This vulnerability arises due to the lack of sanitization in the offset, where it is not restricted to positive values. Consequently, data is not confined to designated spaces after the buffer value, enabling unauthorised access and potential misuse of sensitive information.
To rectify this vulnerability, I implemented an offset validation check. By enforcing strict boundaries and ensuring that the offset remains within permissible ranges, the change will prevent unauthorised access attempts and significantly enhance the robustness of the reference monitor.
Another mechanism that I have fortified is the End Of File (EOF) offset while writing to the file. By first checking for the presence of pending data, the code ensures that it does not inadvertently overwrite existing information. If pending data is identified, the code calculates the total space occupied by this data and compares it with the provided offset. If the provided offset extends beyond the pending data's end, indicating it is past the EOF, the code strategically seeks to this offset and writes an empty string. This operation creates a buffer zone between the existing data and the new offset, safeguarding data integrity. Even in the absence of pending data, the code employs the same strategy, ensuring that the file remains properly organised. In essence, the code's rationale is to maintain a structured file layout, promoting accurate data storage while mitigating the risks associated with unintended data manipulation.

Finally, another function of the program that I reworked in the undo function. The function initially checks whether the file is closed, ensuring that operations are only performed on open files. Subsequently, it verifies if there is committed data available. If there is, the code strategically uses self.LPfile.writeat(self.committed_data, self.pending_offset) to revert the file's content to the state of the last committed data. This action effectively undoes any pending changes, restoring the file to its previous state.