The code is written in **JavaScript** and uses Node.js, as indicated by the `require` statements and the asynchronous filesystem operations.

Regarding the security concerns:

1. **Sensitive Data Exposure**:

   - The code writes sensitive data to a temporary file before encrypting it. This means there's a window of time during which the sensitive data is stored unencrypted on the file system, making it vulnerable to exposure if an attacker gains access to the file system or if other processes read the temporary file.

2. **Potential Race Condition**:

   - After the data is written to the temporary file and before it's deleted, there's a period during which the file is accessible. If another process or attacker knows the timing and filename, they could potentially read the data before it's deleted. This is known as a Time-of-Check to Time-of-Use (TOCTTOU) race condition.

3. **Hardcoded Temporary File Path**:

   - The path `/tmp/sensitive_data.csv` is hardcoded. If multiple instances of this code run simultaneously (e.g., in a multi-user or multi-process environment), they might overwrite each other's files or encounter file access conflicts.

4. **Inadequate Error Handling**:

   - If there's an error in writing or deleting the file, the error is thrown, potentially crashing the application. While it's good to be aware of errors, crashing might not be the desired behavior, especially in production environments.

5. **Use of Synchronous File Operations**:

   - The encryption function `encryptFile` seems to be synchronous (as there's no callback or promise). If the file is large, this might block the Node.js event loop, affecting the application's performance.

### Recommendations:

1. **Avoid Storing Sensitive Data Unencrypted**:

- If possible, encrypt the data in memory before writing it to a file, or avoid writing it to the file system entirely.

2. **Use Unique Temporary Filenames**:

- If you must write to a temporary file, generate a unique filename for each operation to prevent conflicts and reduce the risk associated with hardcoded paths.

3. **Secure Temporary Files**:

- Set appropriate file permissions when creating the temporary file to ensure that only the necessary processes can read or write to it.

4. **Enhanced Error Handling**:

- Consider more graceful error handling strategies, such as logging the error and retrying the operation or notifying an administrator.

5. **Review File Deletion**:

- Ensure the file is securely deleted, and its contents cannot be recovered.

By addressing these concerns, the code's security can be significantly improved.