This system uses a database class to manage and validate all gallery events. The database is written and read as a serialized object through a ciphered stream to the specified log file. Only when provided the original authentication token can the database be accessed.

In detail:

The core of this system is the ‘DB’ or database class. This class maintains a chronological list of gallery events as ‘Entry’ objects, which are composed of several fields such as the timestamp for the event or the room involved. When an Entry is attempted to be added to the DB, it’s fields are validated, or rejected in the case of an invalid event, such as a guest leaving a room they did not most recently enter.

When LogAppend is ran, the arguments are checked for initial validity, such as acceptable combination of arguments or characters in a name. Once the input string is processed, in the case of acceptable input, the specified log file is opened (or created if it doesn’t exist). If there is anything written to the file, an input stream attempts to read it. This will be the most recently written DB object, complete with the list of Entrys, or nothing in the case that this is the first valid entry into the DB.

When the DB is written to the log file after every successful invocation of LogAppend, it is done so through an ObjectOutputStream created from a CipherOutputStream. The ObjectOutputStream writes serializable Java objects, such as DB, in such a way that they can be written and read as that class with no further handling. The CipherOutputStream takes a Cipher object, keyed with a provided encryption key, and performs the specified encryption on objects written through it. In short, the list of events that have taken place at the gallery is written and read as an encrypted DB object.

The Cipher object, utilized by the CipherOutputStream (and in the case of LogRead, CipherInputStream), is keyed with a hash of the user-provided authentication token. The token is hashed via SHA-1 to ensure that it will always be at least 128 bits long for use with AES-128 encryption. The hashed token is used as the encryption key for the Cipher, therefore causing the DB object serialization to be encrypted with this token by the CipherOutputStream, before being written to the log file. As a result, the contents written to the log are indecipherable without the proper authentication token provided for decryption.

After LogAppend has read the most recent version of the DB, an Entry object is attempted to be added to it as described above. In the case that the Entry parameters are valid, the Entry is instantiated and added to the DB’s internal list of events, appropriately named ‘entries.’ If there are more events to process, such as with a multi-entry batch file, those events are also checked and added upon validation. After the last event is processed, the DB is again encrypted and written to the log file as described above.

When LogRead is invoked, the provided arguments are validated in much the same way. If they are found to be acceptable, the DB is read from the specified log with the provided token used for unencryption, using the same methods as LogAppend. The list of entries in the DB is then processed to return the desired information.

Attacks considered:

1. Log file observation
A CipherOutputStream is used to prevent any information from being learned about the events at the gallery through simple observation, such as opening the log for reading. As the DB object is AES-128 (because AES-256 requires some silly extra work to use) encrypted before being written, it has all of the security properties that any other AES-128 ciphertext would, mainly unencryption being practical only with initial knowledge of the encryption key. Relevant lines: LogAppend.java: 323-326, 362-265, LogRead.java: 376-379,

2. Authentication brute-force
The provided authentication token is SHA-1 -hashed to ensure that it is at least 128 bits. This provides brute-force security to logs authenticated with tokens shorter than 128 bits by expanding the key-space for those keys, while maintaining the practical security that a ‘good’ hashing algorithm provides, such as ‘time to break.’ Relevant lines: LogAppend.java: 308-314 , LogRead.java: 365-371

3. Malicious input
Implementation in Java was chosen because it produces ‘managed code,’ preventing attacks such as buffer overflows due to JVM memory management. This is simple, but one of the most important security considerations in design is to use systems that have been thoroughly tested and proven to work.

4. Outside modification
The CipherInputStream automatically provides detection of tampering to the log, as altering the encrypted data will result in failed verification upon reading and attempted unencryption. This prevents undetected modification, and invalidates any logs attacked in this way, as they can not be read or written to. Relevant lines: LogAppend.java: 325, 364, LogRead.java: 378