System Design

Dylan O'Reagan and Will Yeager

In order to ensure the security of the art gallery's logs, our system is designed with three principles in mind - Confidentiality, Integrity and Authentication. We achieve confidentiality by encrypting the log file, integrity by including a hashed record ID before each log entry, and authentication by storing hashed and salted versions of the tokens. This document describes the specific implementation of the system by detailing each of the main components/functions of the two programs, LogAppend and LogRead. Furthermore, it describes four specific attacks that could potentially be carried out against the system, as well as defenses that have been put in place to thwart these attacks.

I.) Log File Format

The plaintext of the log file is stored as a Comma Separated Variable file, with column attributes as follows:

<record-id>,<timestamp>,<A or L>,<room-id>,<guest-name>,<employee-name>

Each record is stored on its own line, and the lines are delimited by the newline character "\n." The log file is encrypted when stored. This encryption is described in section 4.

II.) Command Line Parsing

The first step in our system is parsing the command line options provided to the system to ensure that they do not contain any illegal option combinations. To achieve this goal, we examine the arguments supplied to the program and check them against their constraints. Specifically we ensure the following constraints:

1.) LogAppend

a.) -A and -L options shall not appear together

b.) If -B is one of the options, it takes precedence over all other options, unless it is specified in a batch file

c.) If multiple arguments are supplied, it is the final argument that is used

d.) If the -T option is supplied, the timestamp must be greater than the timestamp of the previous record.

2.) LogRead

a.) -S, -R, -I and -T options shall not appear together

b.) The logfile name shall only appear once

3.) Shared

a.) Room numbers must be non-negative integers between 0 and 1,073,741,823, inclusive

b.) Timestamps must be positive integers between 1 and 1,073,741,823, inclusive

c.) Tokens must contain only alphanumeric characters

d.) Names must contain only alphabetic characters

e.) Only one of either -E or -G shall appear

f.) An argument must be supplied for -E, -G and -K

For both LogAppend and LogRead, if the room number is not specified then -1 is used to indicate the gallery as a whole.

III.) Token Storage and Verification

Only someone who supplies the correct token may decrypt the log file associated with that token. In order to verify that the correct token is supplied, a hash of the token is stored in a file named hashes.txt. This hash provides pre-image resistance of the token, assuming that the adversary was able to view the file used to store the tokens. This file has entries of the form:

<logfile-name>:<token-hash>

Whenever a new command is entered, the supplied token is hashed and compared against the hashed token in hashes.txt. If the tokens do not match then “integrity violation” is printed and the program exits. The tokens are hashed using the jBCrypt library, with 16-bytes of randomly generated salt.

IV.) Log Encryption

We achieve privacy/confidentiality of the Log File by encrypting it using a key stored in a file of the form <logfile>.key. This key is used in tandem with an initialization vector stored in a file of the form <logfile>.iv. These files are generated the first time that

the logfile is created. The initialization vector is seeded with randomness using the SecureRandom class of the java.crypto package, and the key is generated using the KeyGenerator class of the same package.The encryption scheme used is AES-128 in Cipher Block Chaining mode with PKCS5Padding, to ensure the appropriate block size. Our system assumes that the user will store these files in a private directory, separate from the log file; although, for the purposes of this competition they are stored in the same directory.

When encrypting the log records, we encrypt the entire file as a whole. Since we do not encrypt the log records separately, there is no issue of ciphertext appearing as a delimiter between records. Whenever the log needs to be appended to or read from, the entire log file is read into memory and decrypted for processing.

V.) Record Validation

We achieve integrity by supplying a record ID for each line of the log file. This hash allows our system to alert a user as to whether or not the log has been tampered with. The recordID is created by concatenating the line number of the record currently being inserted with the previous line's record ID then hashing with Java's String.hashCode method. The record ID of the first line is generated by hashing the number "1" concatenated with the path of the log file. This setup ensures that an adversary cannot insert a record into the middle of a log file, as its hash will not match that of the preceding record.

VI.) Log Appending/Reading

When accessing the log, the first thing that is done is to verify the hash of the supplied token, as described in Section 3. After decrypting the file using the key associated with the supplied token, as described in Section 4, the file is processed line-by-line to ensure it has not been tampered with. This consists of hashing the line number with the record-id of the preceding line, and ensuring that they match. While this is being done, the timestamp of each record is checked to ensure that there were no erroneous or malicious additions to the file. Additionally, the state of the current user's location in the art gallery is updated as records pertaining to that user are processed. When the end of the file is reached, the record to be appended is compared to the current state of the user under review. If the timestamp, location, and action of the record to be appended don't conflict with the current state of the user, then the record is appended to the plaintext of the log. This plaintext is then encrypted and written back to

the original log file. However, before the encrypted text is written, the ciphertext is hashed using Java's built in hashing method for strings. This hash is then stored in a file of the form <logfile>.hash in order to be checked against the ciphertext when reading the log, and provides message authentication to ensure the log has not been tampered with.

The process of reading from the log is the same as appending to the log. The only difference is the processing of the information that is done when iterating through the log.

Four Specific Attacks

Security Model:

Without knowledge of the token an attacker should *not* be able to:

- Query the logs via logread or otherwise learn facts about the names of guests, employees, room numbers, or times by inspecting the log itself
- Modify the log via logappend.
- Fool logread or logappend into accepting a bogus file. In particular, modifications made to the log by means other than correct use of logappend should be detected by (subsequent calls to) logread or logappend when the correct token is supplied

Brute Force Attack

Attack:

Knowing that it is quite possible that the log owner used a short password for their log file, the adversary could attempt to perform a brute force attack on the token supplied to the program. In this attack, the adversary would simply run through every combination of arbitrary length tokens up to a certain length in the hopes that one of them would match the hash of token used to create the log file. To do this, the adversary would simply have to try to read the log using every potential token value until one of them matched the value used for the log file.

Defense:

We defend against this attack in two different ways. The first defense that we implement is the use of 16-bytes of salt when hashing the tokens supplied to the LogAppend program. This added randomness makes the chance that the adversary will find a collision exponentially smaller. It also ensures that even if the adversary is able to find a collision, then the string used will be a hash of both the token and the salt - not just the token itself.

The second defense against this attack that we implemented is the use of the BCrypt library when hashing tokens. BCrypt is by design a very slow hashing algorithm. This limits the speed at which the adversary can supply attempts to the LogAppend or LogRead programs, and greatly increases the time necessary in order to find a collision.

This defense is implemented on lines 316 and 595 of the LogAppender.java file in the LogAppend source code, and line 27 and 408 of the Main.java file in the LogRead source code.

Invalidation through Prepending of Records

Attack:

An adversary could prepend a fake record to the beginning of log to invalidate the rest of it. They could either have the first record have a timestamp of the maximum value, or claim that a guest or employee has arrived at or left a room illegally (e.g. leaving before they've arrived). This invalidates the log, and if processed would supply the user with false information about the state of the art gallery.

Defense:

Our use of Record IDs and cipher streaming ensures that we detect any illegal prepending/appending to the log. This is described in depth above in section 4, but in short we set the record id of each line to be the hash of the line number concatenated with the previous record id. This ensures that if the log were tampered with we would be able to detect it.

Another defense that protects against this attack is the encryption of the log file itself. This means that if an attacker could find a way to append to the log, but not decrypt it, then what they append would be gibberish after the log gets decrypted. While

they would have managed to invalidate the log, their efforts would be detected by the check to the record ids which would return “integrity violation” the next time the log was used.

Initially we had implemented the hashing and checking of the record IDs using the jBcrypt library, but this resulted in our code timing out. After reading into the issue further we discovered that this is by design. jBcrypt is written to always take a significant amount of time when hashing to make brute force attacks more difficult. Unfortunately, this meant that if the log had any more than 5 or 6 lines it ended up taking a few seconds to either append or read, since every line’s record ID had to be checked against the previous line’s. We solved this by using a slightly less secure but much more efficient hashing algorithm: String.hashCode.

This defense is implemented on lines 86, 246, 325 and 565 of the Main.java file in the LogRead source code. In the LogAppend source code this defense is implemented on lines 202-204 and lines 271-276 of the LogAppender.java file.

Predictable IV Attack

Attack:

If a predictable initialization vector is used when encrypting the contents of the file, then an adversary can potentially learn information about the plaintext of the file by using the ciphertext of the log file to encrypt a crafted plaintext input. If the attacker creates a record with fields the same as those of the one of the records in the log file, then they can exploit the way that Cipher Block Chaining encrypts data in order to determine whether or not their guess is correct. By crafting their plaintext such that it equals a potential value of an existing record XORed with the existing records IV and the ciphertext of the existing record, the adversary makes it so that encrypting the plaintext will return the same ciphertext as the previous block. This is due to the fact that CBC XOR’s the ciphertext of the previous block with the plaintext. When the malicious plaintext is XORed with the previous ciphertext, what gets encrypted ends up being the same value as the existing record’s plaintext XORed with its initialization vector. This will necessarily encrypt to the same value ciphertext as the existing record, revealing whether the attacker’s guess was correct.

Defense:

In order to defend against this attack, the initialization vector used to encrypt the log file is changed each time that the file is encrypted. This prevents the attacker from

guessing the value of the IV used to decrypt the existing record, and makes it so that they cannot craft a plaintext which will encrypt to the same value as the existing record. Additionally, the initialization vector must be changed in a random fashion so that the adversary cannot guess the IV to be used in the next encryption. To accomplish this, we seed the initialization vector with a random value using Java's SecureRandom class each time the file is encrypted, and store the new initialization vector in a file of the form <logfile>.iv. This creates a random initialization vector that cannot be guessed, and prevents the adversary from being able to create an appropriate plaintext.

This defense is implemented on lines 447-464 and line 644 of the LogAppender.java file in the LogRead source code.

Padding Oracle Attack

Attack:

Since our system uses Cipher Block Chaining to encrypt the log contents, it is susceptible to a padding oracle attack. This could be accomplished by modifying the bytes near the end of the file such that they represent a guess as to the files real contents XORed with a known value used for padding. This padding value can be determined from the padding scheme used in the encryption of the file. Then, when the file is decrypted, the byte that was modified will appear as proper padding in the plaintext and the oracle will return without any errors. The modified byte should be created such that it represents the guess of the next plaintext byte XORed with the original ciphertext byte and the padding value. When the oracle decrypts the block the ciphertext and original plaintext will cancel out, leaving the padding value as the plaintext.

Defense:

In order to defend against padding oracle attacks, we use a simple message authentication to ensure that the ciphertext of the log file has not been tampered with. Before writing the ciphertext to the log file, we hash it using Java's built-in string hashing function and save the hash to a file named <logfile>.hash. Then, when reading from the log file, we compare the hash of the log file with the stored hash generated when the file

is saved. If the hashes don’t match then we can alert the user that the file has been tampered with, and that they may be victim to a padding oracle attack.

This defense is implemented on lines 646-655 of the LogAppender.java file in the LogAppend source code and lines 633-657 of the Main.java file in the LogRead source code.