CS 419 Project 2: Tamper-Evident Logging

Due Sunday, March 31, 2024 11:59pm

Introduction

Event and error logging play a crucial role in the management and security of computer systems for several reasons:

1. **Troubleshooting and Performance Monitoring**: Logs provide detailed information about the system's operations, including errors and exceptions. This information is invaluable for diagnosing issues, optimizing system performance, and understanding how the system is being used.

2. **Security and Compliance**: Logging is essential for detecting unauthorized access, monitoring suspicious activities, and ensuring compliance with regulatory standards. It helps identify potential security breaches and mitigate them in a timely manner.

3. **Audit Trails**: Logs serve as an audit trail, allowing administrators to review historical data for changes, access patterns, and transactions. This is critical for forensic analysis in the event of a security incident or compliance audit.

Attackers are aware of the importance of logs in detecting and analyzing their activities. Therefore, one of the first actions an attacker might take after gaining access to a system is to delete or tamper with its logs. This erases any evidence of their entry and activities, making it difficult for administrators to trace the breach and understand the extent of the compromise.

To counteract such tactics, sending logs to a remote server is recommended. This practice, known as log forwarding or centralized logging, ensures that copies of the logs are stored in a secure location separate from the local system. Even if an attacker manages to alter or delete logs on the compromised system, the remote copies remain intact for investigation.

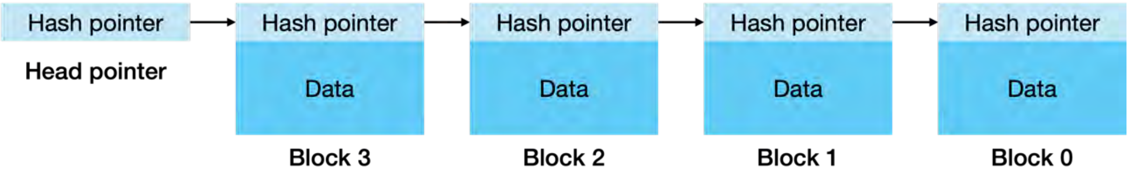
Furthermore, a more insidious threat comes from attackers or even untrustworthy administrators who may attempt to modify log entries or selectively delete them. This can be more damaging as it allows malicious activities to go unnoticed or makes the forensic analysis more challenging by presenting a misleading or incomplete picture of the events.

To mitigate these risks, organizations should employ secure, tamper-evident logging mechanisms. This includes using cryptographic techniques to sign logs, ensuring their integrity, and employing robust access controls and monitoring to detect and prevent unauthorized log manipulation.

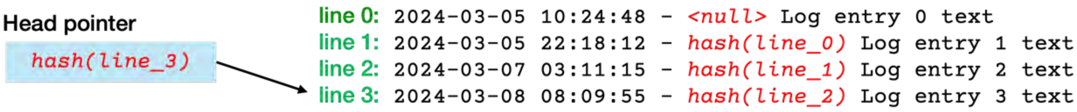
Your assignment

Your assignment is to construct a basic blockchain-based logging service. You will implement a list of log entries linked via hash pointers and storing the head of the list in a secure file.

The premise of a hash pointer is that a pointer in each block of data contains a reference to the next block in the list along with a hash of the contents of that next block. Those contents include the block pointer in that block. For example, the hash pointer of Block 2 is a reference to Block 1 along with the hash of Block 1.



In a log file, each log entry is a single line. A block is simply a line in the file. The “reference” to the previous block is simply its position. The previous block of a log entry is simply the previous line. Hence, a hash pointer is simply the hash of the entire previous line of text.



```
line 0: 2024-03-05 10:24:48 - <null> Log entry 0 text
line 1: 2024-03-05 22:18:12 - hash(line_0) Log entry 1 text
line 2: 2024-03-07 03:11:15 - hash(line_1) Log entry 2 text
line 3: 2024-03-08 08:09:55 - hash(line_2) Log entry 3 text
```

If we’re not implementing a decentralized blockchain, then there is no need to create a system that requires using more computing power than Fredonia's entire electrical consumption to compute a proof of work. We can maintain a hash pointer to the head of the list and store it in a trusted location. For a log file, this is simply the hash value of the last element of the list.

Why a blockchain?

Simply adding a hash of the message to each line of the log would enable us to check if the data on that line has been corrupted. However, it would not provide a mechanism to detect whether any lines are missing. Also, if each line contains a hash for the message on that line, an attacker would be able to simply replace the message and compute a replacement hash for the new message.

A list of hash pointers allows us to detect whether data has been corrupted and whether any lines have been removed.

What to write: program specifications

You will write two programs: `addlog` and checklog.

`addlog`

The `addlog` program adds a timestamped log entry to the log file containing a user-supplied string. The usage of the command is:

```
addlog log_string
```

The program should print a usage message and exit gracefully if no arguments are supplied or if more than one argument is provided to the command.

The log file is named `log.txt` and is stored in the current working directory. You can open it without specifying a path. The head of the list is stored in a file called `loghead.txt`. Neither file exists initially, and you must create them if they are missing:

- If `log.txt` is missing, you will ignore `loghead.txt` if the file exists, create a new log file, and then create `loghead.txt` to contain the hash of the newly-added log entry.
- If `loghead.txt` is missing but `log.txt` exists, you have no way of knowing whether the last lines of the log file were deleted or if new entries were appended. While you can recreate a new head pointer, it is sufficient to simply print an error message stating that the file is missing and exit the program.

To add an entry to a log:

1. Read the hash from the head pointer file (`loghead.txt`). This is the string representation of the hash of the last line of the file. If the file is empty and the log file (`log.txt`) does not exist, the head pointer file is initialized with the string `begin`.
2. Create a log entry string containing the timestamp, the hash just read from the file (or `begin`), and the log text provided by the user.
3. Append the log string in step 2 to the log file.
4. Create a base-64 encoded SHA-256 hash of the complete log string from step 2 (without a terminating newline).
5. Replace the hash in the head pointer file with the hash computed in step 4.

Log format

Each log line will contain a timestamp, a hyphen separator (`‘-‘`), an encoded hash string, and the log text. Each of these will be separated by spaces (`‘ ‘`) and terminated by a newline (`‘\n‘`). The elements are as follows:

1. **Timestamp format**
   The timestamp can be in any reasonable format that expresses the year, month, date, hour, minue, and second. There is no need for sub-second granularity and you don’t need to bother with representing time zones. For example, you may choose to use an [ISO 8601](ISO 8601) format. For example:
   ```
   2024-03-12T13:14:55Z
   ```
   Or a format typically found in Linux syslog entries, such as:
   ```
   Mar  12 20:52:07
   ```
   Or any other reasonable and readable format.

2. **A hyphen**
   The hyphen (‘`-`‘ or 0x2d) simly serves to act as an easy-to-parse way of getting past the date string to allow the timestamp to contain spaces.

3. **A base64-encoded SHA-256 hash**
   Each log entry contains a hash of the previous line (the entire string, including the timestamp but not the terminating newline). A 256-bit SHA-2 hash (referred to as SHA-256) will be generated, and the 256-bit output will be encoded in a base64 format.

   Base64 encoding is a common technique used to convert binary data into a text format. It works by dividing the data into 6-bit blocks, each of which is then mapped to a specific character in a 64-character alphabet, resulting in a text representation that can be handled by systems designed primarily for text data.

   For example, the hash of the string:
   ```
   2024-03-05 10:24:48 - begin Log entry 0 text
   ```
   can be expressed as this hexadecimal string:
   ```
   eeb696e6227feadc158cc986bde75d124b0cdab051f864ebe7e5ad7ee9c36482
   ```
   which corresponds to this base64 string:
   ```
   7raW5iJ/6twVjMmGveddEksM2rBR+GTr5+WtfunDZII=
   ```

4. The user-provided string
   This is the string provided as the first argument to the `addlog` command. The string cannot contain newline characters. Any newline characters provided by the user must be converted to spaces.

A sample log file might look like this:

```
2024-03-05 10:24:48 - begin Log entry 0 text
2024-03-05 22:18:12 - 7raW5iJ/6twVjMmGveddEksM2rBR+GTr5+WtfunDZII= Log entry 1 text
2024-03-07 03:11:15 - x8oTYT1K3jYwoR1E9APptURkMGYWCWDHHlE0R8s/tYU= Log entry 2 text
2024-03-08 08:09:55 - T0hGjcooesiJZ/grkOixDrXJQokPZ1yPHX8HkbZNmWw= Log entry 3 text
```

With loghead.txt containing the base64-encoded SHA-256 hash of the string

```
 2024-03-08 08:09:55 - T0hGjcooesiJZ/grkOixDrXJQokPZ1yPHX8HkbZNmWw= Log entry 3 text
```

Which is:

```
        C0QJxdWrVozmOOqmjfl7f/cF8HPUCmKYAqssiBfoR10=
```

checklog

The `checklog` program scans the log file to validate its integrity. The usage of the command is:

```
checklog
```

To validate the integrity of the log, start at the beginning of the log file:

1. Read a line *L*. Check that the hash value on the first line is the string `begin`.
2. Compute the base-64 encoded SHA-256 hash of *L*. $H = b64encode(sha256(L))$.
3. Read the next line and parse out the hash. If the line cannot be read because you reached the end of the file, read the hash stored in the head pointer file (`loghead.txt`).
4. Compare the hash extracted in step 3 with *H*.
5. If they do not match, report the line number of the corruption (the previous line) and exit.
6. If you didn't reach the end of the file, go to step 2 and continue until there are no more lines to read.

You can use other steps to verify the integrity of the log. For instance, you can read the entire file and the head hash pointer into memory. Then, starting from the head and working from the last line to the first, check that the current hash matches the computed hash of the previous line.

If the validation succeeds, then print the string:

`valid`

and exit the program with an exit code of 0.

If the validation fails, including for the reason that either the log file or head file are missing, print the string

`failed:` ***`error_message`***

where ***`error_message`*** is a message describing the failure and exit the program with an exit code of 1.

If the head pointer file is missing, you should report that and exit.
If the log file is missing, you should also report that and exit.

Testing

Some tests you should perform before submitting your program include the following:

`addlog`

1. Create `loghead.txt` and `log.txt` when they don't exist.
2. Print a message if no arguments or more than 1 argument are present and exit gracefully.
3. Any newlines in the command-line argument should be converted to spaces.
4. An empty string can be accepted as valid log entry.

`checklog`

5. Print an error message if any command-line arguments are provided and exit.
6. Detect that `loghead.txt` is missing.
7. Detect that `log.txt` is missing.
8. Print a `valid` message if log file validation succeeds.
9. Delete a line of the file. A `failed` message should be generated, identifying the preceding line number (line numbers start with 1 by convention).
10. Modify the text or timestamp of a line of the file. A `failed` message should be generated, identifying the line number of the modified line.
11. Modify a hash in a line of the file. A `failed` message should be generated, identifying the line number of the preceding line.
12. Delete the first line of the file. Validation should fail, identifying the lack of a starting line.
13. Modify the head pointer file. Validation should fail on the last line of the file.

14. Under no circumstances should either program produce runtime exception, stack trace, or core dump.

What to submit

Place your source code into a single *zip* file. If code needs to be compiled (i.e., Java, C, or Go), please include a `Makefile` that will create the necessary executables.

We don't want to figure out how to compile or run your program. We expect to:

1. unzip your submission
2. run make if there's a Makefile to build everything that needs to be built
3. Set the mode of the programs to executable:
   ```
   chmod u+x addlog checklog
   ```
4. Run the commands as:
   ```
   ./addlog "this is a test of a log entry"
   ./checklog
   ```

If you are using python, you can submit either:

(a) A shell script with the name of each command that runs each program. For example, a file named `addlog` that contains the text:

```
#!/bin/bash
/usr/bin/python3 ./addlog.py “$@”
```

or

(b) (*preferably*) a program named `addlog` (*not* addlog.py) that contains your source code and starts with the line:

```
#!/usr/bin/python3
```

You will do the same for checklog.

If you are using Java, you will have a simple `makefile` that compiles the Java code to produce class files. The `addlog` program will be a script that runs the *java* command with the necessary arguments, containing contents such as:

```
#!/bin/bash
CLASSPATH=. java Addlog "$@"
```

If you're using C, Go, or any language available on the iLab machines, provide a `makefile` for generating the required executables.

Test your programs and scripts on an iLab machine to make sure they work prior to submitting.