CS 419 Project 2:
Remote Tamper Evident Logging With Proof-of-Work

Due Sunday, March 16, 2025 11:59pm

Introduction

Event and error logging plays 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 promptly.

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 know 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 modify 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, it is recommended that logs be sent to a remote server. 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.

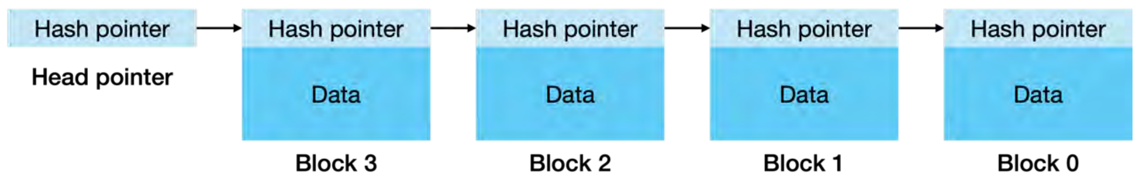
Another more insidious threat comes from attackers or even untrustworthy administrators who may attempt to modify log entries or selectively delete them. This can be more damaging as it allows malicious activities to go unnoticed or makes the forensic analysis more challenging by presenting a misleading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

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

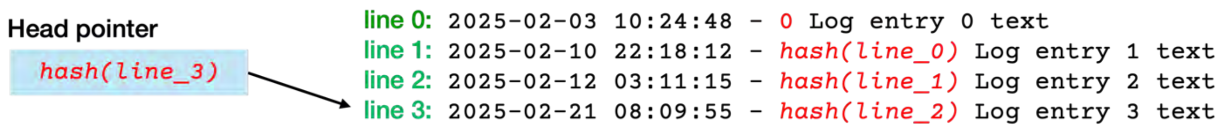
Your assignment

Your assignment is to construct a blockchain-based logging service. You will implement a list of log entries linked via hash pointers and store 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 in the log file. 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: 2025-02-03 10:24:48 - 0 Log entry 0 text
line 1: 2025-02-10 22:18:12 - hash(line_0) Log entry 1 text
line 2: 2025-02-12 03:11:15 - hash(line_1) Log entry 2 text
line 3: 2025-02-21 08:09:55 - hash(line_2) Log entry 3 text
```

Since we're not implementing a decentralized blockchain, we can keep a hash pointer to the head of the list and store it in a secure location. For a log file, this is simply the hash value of the last element of the list. We only need to protect this element to allow us to validate the integrity of the entire list.

With a decentralized blockchain, there is no trustworthy head pointer, so an immensely challenging proof of work computation makes it incredibly difficult to modify history and update all subsequent blocks to create the longest blockchain. Because of this, the longest chain is considered the authoritative one in such a system.

Why a blockchain?

Simply adding a message hash 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 could 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.

Proof of Work

The concept of Proof of Work used in Bitcoin came from an invention called hashcash. The idea was to make it difficult for a spammer to spend vast amounts of email because each message would contain proof of work that is a function of the message content, including the recipient's address. Generating the proof of work would take significant computation, reasonable for sending a few messages but prohibitive for sending millions.

We will adopt the same principle here to make it difficult for a program to flood a log with a huge number of messages in a short time. Every message sent to the logging server will contain a proof of work string that results in the SHA-256 hash of the combined message containing 22 leading zero bits. The server will reject any message that does not meet this requirement.

For example, consider the string `"This is the first message in the log"`.

We will now compute the proof of work as a string such that the complete message containing this value followed by a "`:`" and followed by the original message will result in a string whose 256-bit SHA-2 hash will start with at least 22 bits that are 0. There are many possible strings, and we need to iterate through various combinations of printable characters to find something that works. I recommend sticking with the set `A-Za-z0-9`, or at least ensuring you're generating printable output and don't use ":". For this example, we discover "`xy4m`". Let's run the *openssl* command on the new message:

```
$ echo -n 'xy4m:This is the first message in the log'|openssl dgst -sha256
SHA2-256(stdin)= 0000035cac269f60ebeec9fff39bc65ff1b65b81aed4bb68888ec1521afe7f0d
```

The leading bytes of the hash are `0000035c`, which convert to binary as:

```
      0000035c = 0000 0000 0000 0000 0000 0011 0101 1100
```

This hash contains 22 leading zero bits, which satisfies our condition that the first 22 bits of the hash are zero.

What to write: program specifications

You will write three programs: `log`, `logserver`, and `checklog`.

log

The `log` program takes an arbitrary string from the command line and sends it to the log server. The program's usage is:

```
log server_port_number message
```

where the *server_port_number* is the port number on which the log server is listening for connections and is printed when you start the server. For this assignment, we assume the server is running on the same host ("`localhost`"). In a real deployment, you would change this to the actual hostname where the service is running and use a standard port number.

If two arguments are not supplied to the command, the program should print a usage message and exit gracefully. The *message* can contain arbitrary printable text, including spaces.

The program will:

1. Convert whitespace in the message into space characters (i.e., replace newlines and tabs with spaces). You may write your `log` and `logserver` programs with the assumption that messages will not be longer than 256 bytes.
2. Generate a proof-of-work string such that the SHA-256 hash of the concatenation of
   proof-of-work + ":" + message
   is a value where the first 22 bits are 0. This is an exhaustive process, and don't be surprised if it takes hundreds of thousands of iterations of testing various single-character, two-character, three-character, etc. strings, moving on to longer strings until you find a hash with the property you need. For example, the message "`My favorite class is physics.`" may result in the string:
   ```
   "HCTi:My favorite class is physics."
   ```
3. Send the resulting string to the server.
4. Wait for a response from the server and print the response string.

`logserver`

The `logserver` program runs forever, listening for connections from log programs. It takes no arguments and starts by printing the port number on which it is listening for connections.

Upon accepting a connection from the `log` program, it receives a single line of text. It validates the message to ensure that the first 22 bits of the SHA-256 hash of the message (without a terminating newline) are all 0.

It then strips off the text up to and including the "`:`" character since this was just the proof-of-work value and is no longer needed. If the "`:`" character is missing, then the string is considered invalid.

It adds a timestamped log entry to the log file containing the message.

The log file is named `log.txt` and is stored in the current working directory. You can open it without specifying a path. The hash of 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 (or overwrite) `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 if the last lines of the log file were deleted or if new entries were appended. While you can recreate a new head pointer, you can no longer trust the integrity of the log, so you should return an error message stating that the file is missing and not log the message.

To add an entry to a log:

1. Read the hash value 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 or does not exist and the log file (`log.txt`) does not exist:
   The head pointer is initialized to the string "`start`" if the log file (`log.txt`) does not exist.
   b. An error stating that the head pointer is missing is sent back if the log file does exist.
2. Create a log entry string containing the timestamp, the hash just read from the file (or the string "`start`" – without the quotes), and the log text provided by the user (without the proof-of-work).
3. Append the log message string in step 2 to the log file.
4. We want the hash to be printable and, for this assignment, a relatively short string. Create a base64 encoded representation of the SHA-256 hash of the complete log string from step 2 (without a terminating newline) and use the ***last 24 bytes*** of the result. See the discussion below on generating the hash.
5. Replace the hash in the head pointer file (`loghead.txt`) with the hash computed in step 4.

The `logserver` program sends a single newline-terminated line of text with either an "`ok`" or an error message back to the client.

Log format

Each log line will contain a timestamp, a hyphen separator (`'-'`), an encoded hash string, and the log text. These items will be separated with spaces (`' '`) and the entire line will be 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, minute, 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 format. For example:
   ```
   2025-12-12T13:14:55Z
   ```
   Or a format typically found in Linux syslog entries, such as:
   ```
   Feb 12 20:52:07
   ```
   Or any other reasonable and readable format.

2. **A hyphen**
   The hyphen (`'-'` or 0x2d) after the timestamp simply serves as an easy-to-parse way of getting past the date string and allowing the timestamp to contain spaces.

3. **A base64-encoding of 128-bits of a 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 (commonly referred to as SHA-256) will be generated.

   For this assignment, we want the hash to not take up a big chunk of the log entry, so we will sacrifice some cryptographic security by using a 128-bit (16-byte) hash instead of a 256-bit hash. We'll create this simply by using the last 24 bits of a base64 encoding of the hash.

   Base64 encoding is a common technique for converting binary data into text. It divides the data into 6-bit blocks, each of which is mapped to a specific character in a 64-character alphabet. This results in a text representation that can be handled by software designed primarily for printable text data.

   For example, the hash of the string:
   ```
   2025-02-23 07:17:24 - start first message
   ```

   can be expressed as this hexadecimal string:
   ```
   4d7e3a08f6cfa7060391bc62a1a897743ead05ea5fd16bfd859df2a5d72f3614
   ```

   You can run the command
   ```
   echo -n '2025-02-23 07:17:24 - start first message'|openssl dgst -sha256
   ```
   to test this.

   The base64 encoding of the binary hash (not the hex string) is:
   ```
   TX46CPbPpwYDkbxioaiXdD6tBepf0Wv9hZ3ypdcvNhQ=
   ```

   And the last 24 characters are:
   ```
   dD6tBepf0Wv9hZ3ypdcvNhQ=
   ```

   Which is the value we store in loghead.txt when we add this log entry to `log.txt`. The next log entry will contain this value since it is the hash of the previous line in the log.

4. **The user-provided string for logging**
   This is the string provided as the argument to the `log` command. The string cannot contain newline characters. Any whitespace characters in the message should have been converted to spaces by the `log` program.

A sample log file might look like this:

```
2025-02-23 07:17:24 - start first message
2025-02-23 07:19:31 - dD6tBepf0Wv9hZ3ypdcvNhQ= message #2
2025-02-23 07:19:43 - IhxNvaB5ToBU+vyozmn4NZQ= Log entry 1 text
2025-02-23 07:20:01 - pVOA5RnFAP00ArVcZOc6EZk= Log entry 2 text
2025-02-23 07:20:18 - NGii2r+NVKd+hlVBv8jevjY= Fifth log message
```

The file `loghead.txt` contains the last 24 characters of the base64-encoding of the SHA-256 hash of the last line in the log:

```
2025-02-23 07:20:18 - NGii2r+NVKd+hlVBv8jevjY= Fifth log message
```

which is:

```
kKG2s2DJucWqiddUzAMH/dI=
```

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 `start`.
2. Compute the base64 encoded representation of the last 24 characters of the SHA-256 hash of *L*:
   H = last_24_chars(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 because 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.

What you are given

You do not need to write any networking code for this assignment.
You are provided with two skeletal programs that do the basic network tasks:

1. `logserver` - This is the logging server. When it starts, it prints the port number on which it's listening, which will be used as the first argument on the client command. It then waits for client connections. When it receives a client message, it appends it to a file and sends back a return message.
2. `log` - This is the client logging program. It is given the port number and the message to be logged as parameters on the command line. It connects to the server (localhost on the specified port number), sends the message, waits for a response string, and prints the response.

There are versions of these programs in Java, Python, and C, each under a directory of that name. You may use any of these as a template, depending on your choice of language. You can build them by running the *make* command in the appropriate directory.

You will need to expand these programs to do the tasks required in the assignment.

In the *log* program, you will have to convert newlines to spaces and generate a proof-of-work string before the message is sent to the server.

In the *logserver* progam, you will need to validate the hash of the received message, strip off the proof-of-work, create a log entry that contains the timestamp, the previous hash, and the message, append it to the file, and update `loghead.txt`.

I also included a version of a `b64hash` program in each directory that prints a base64 encoded string of a SHA-256 hash of text supplied on the command line.

Reference programs

I've provided reference executable programs for `log` and `logserver` that you can use to test that your interfaces work. Your proof-of-work values may end up being different than the ones I compute, but they should generate hashes with the required properties. Your timestamp format may also differ.

The `log` reference program takes an optional command line argument to change the proof-of-work computation so you can easily try strings that result in invalid values. The `-b` parameter lets you specify the minimum number of leading zero bits in the SHA-256 hash for the message when doing its proof-of-work calculation. The default value is 22. For example, the command

```
log -b 12 50555 "this is a test"
```

will create a proof of work string so the hash will contain at least 12 leading zero bits (e.g., `"n6B:this is a test"`) and send it to the server, which should reject it. Note that the algorithm only tests that the resulting hash will contain *at least* that many leading zero bits, and it's possible that certain messages may happen to result in more zero bits than the minimum requested, although this will be unlikely for a broad set of messages.

Be sure to run the `logserver` and `log` commands on the same machine since the `log` program is hard-coded to connect to `"localhost"`, which resolves to the address of the local network interface.

The programs are under the `test_programs` directory, which contains three subdirectories:

- `linux-aarch64` – Linux versions for the Raspberry Pi
- `linux-x86_64` – Linux versions for Intel platforms (the iLab environment)
- `macos-arm64` – macOS versions for M-series CPUs (but you'll have to override macOS's protections)

Testing

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

`log`

1. Print a message if two arguments are not present on the command line and exit gracefully.
2. Any newlines in the command-line argument should be converted to spaces.

`logserver`

1. Create `loghead.txt` and `log.txt` when they don't exist.
2. Validate the hash of the received message to ensure it starts with at least 22 zero bits.
3. Generate a valid timestamp.
4. Strip off the proof-of-work part.

`checklog`

1. Print an error message if any command-line arguments are provided and exit.
2. Detect if `loghead.txt` is missing.
3. Detect if `log.txt` is missing.
4. Print a `valid` message if log file validation succeeds.

5. Delete a line of the file. A `failed` message should be generated, identifying the preceding line number (line numbers start with 1 by convention).
6. 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.
7. Modify a hash in a line of the file. A `failed` message should be generated, identifying the line number of the preceding line.
8. Delete the first line of the file. Validation should fail, identifying the lack of a starting line.
9. Modify the head pointer file. Validation should fail on the last line of the file.

10. Under no circumstances should any of your programs produce a runtime exception, stack trace, or core dump. Catch any possible errors and print human-friendly error messages.

Performance

Generating a proof of work is compute-intensive, but shouldn't take more than a few seconds per log entry given the requirements in this assignment. If you find that the proof of work is taking a long time (e.g., more than 10 seconds on average), then you need to reconsider your approach and reimplement the proof of work operation.

What to submit

Place your source code into a single *zip* file. If code needs to be compiled (i.e., Java, C, or Go), create or modify a `Makefile` to 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 log logserver checklog`
4. Start a server as:
   ```
   ./logserver
   ```
   The logserver will tell you what port it's listening on. Use that for the log commands in the next step. We're using 50555 as an example.
5. In another windo on the *same* computer, run the commands as:
   ```
   ./log 50555 "this is a test of a log entry"
   ./checklog
   ```

If you are using python, create programs named `log`, `logserver`, and `checklog` (*not* `checklog.py`) that contain your source code and start with the line:

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

When execute permissions are set for the file (see step 3, above), we will be able to run it as the command without having to explicitly invoke the `python` interpreter.

If you are using Java, you will have a simple `Makefile` that compiles the Java code to produce class files. The `log`, `logserver`, and `checklog` programs will be scripts that run the *java* command with the necessary arguments, containing content such as:

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

You can copy these from the `Makefile` under the `java` directory.

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 before submitting.