Vault Hacking Race: A Simple Parallel Computing Exercise to Learn the Concepts of Threads

Mayas Alkhateeb

November 2025

1 Introduction

This report documents the implementation of the ”Vault Hacking Race” assignment, a simple parallel computing exercise designed to illustrate multi-threading concepts in Java. The project simulates a race where multiple hacker threads attempt to crack a vault's password (a random number between 0 and 9999) while a police thread counts down a 10-second timer. If a hacker succeeds before the timer expires, they ”win”; otherwise, the police ”catch” them.

The assignment, provided in the course materials from the Higher Institute for Applied Sciences and Technology (HIAST), focuses on key threading principles such as concurrent execution, shared resources, thread priorities, and termination. The implementation includes both a console-based version for basic demonstration and a Swing GUI for visual progress tracking.

The source code is available on GitHub: `https://github.com/k3rnel-paN1c5/VaultHackingRace`.

2 Assignment Requirements

The assignment requires implementing the following components:

- **Vault Class**: Holds a random password (0-9999) and provides a method `isCorrectPassword(int guess)` that simulates a 5ms delay for each guess to mimic processing time.
- **Ascending Hacker Thread**: Guesses passwords sequentially from 0 to 9999.
- **Descending Hacker Thread**: Guesses passwords sequentially from 9999 to 0.
- **Police Thread**: Counts down from 10 seconds (1 second per tick) and ends the program with ”Game over for you hackers” if it reaches zero first.
- **Main Program**: Creates the vault with a random password, starts all threads simultaneously, and sets maximum priority for hacker threads.

Expected behavior:

- Hackers run concurrently, sharing the vault resource.
- The program terminates when a hacker finds the password or the police timer expires.

3 Implementation Details

The project extends the basic requirements with additional features like a random hacker, an optional binary search hacker, and a GUI for visualization, while maintaining focus on threading concepts.

3.1 Code Structure

The project is organized as follows:

- **model/**: Contains core classes for the vault and threads.
- **ui/**: Contains the Swing GUI.
- **Main.java**: Entry point to launch the GUI (or console version).
- **ConsoleMain.java**: Launches the console version.

Key classes:

- **Vault.java**: Manages the password and guess verification with a 5ms sleep.
- **HackerThread.java**: Abstract base class for hackers, enforcing a `guessPassword()` strategy and providing progress tracking.
- **AscendingHackerThread.java**: Implements ascending guesses.
- **DescendingHackerThread.java**: Implements descending guesses, with progress normalized to ascending scale.
- **RandomHackerThread.java**: Uses random unique guesses via a HashSet to avoid duplicates.
- **BinarySearchHackerThread.java**: (Optional) Uses binary search for efficiency, with a custom `comparePassword(int guess)` method in Vault.
- **PoliceThread.java**: Countdown timer with 1-second sleeps.
- **VaultHackingGUI.java**: Swing interface with progress bars for hackers and police, start/restart buttons, and winner display.

Threads share the `Vault` instance implicitly synchronized by the delay in guess verification. Hackers run at `Thread.MAX_PRIORITY` for fair scheduling. Program termination uses `System.exit(0)` for simplicity in this educational demo.

3.2 Threading Concepts Demonstrated

- **Concurrency**: Multiple hackers guess simultaneously, showcasing how threads can perform independent tasks in parallel.
- **Shared Resources**: All hackers access the same `Vault`, with the 5ms delay simulating contention.
- **Priorities**: Setting high priority for hackers ensures they get more CPU time compared to the police thread.
- **Interruption and Termination**: Threads check for game-over conditions (via GUI or direct checks) to stop early.
- **Synchronization**: Implicit through method calls; no explicit locks needed due to the simple shared state.

3.3 Extensions and Enhancements

- Added a **Random Hacker** to demonstrate non-deterministic threading.
- Implemented an optional **Binary Search Hacker** to contrast efficiency in a multi-threaded context (commented out in GUI for basic focus).
- Developed a **Swing GUI** with progress bars, making thread progress visually observable.
- Included console output for real-time updates in non-GUI mode.

4 How to Run

1. Clone the repository: `git clone https://github.com/k3rnel-paN1c5/VaultHackingRace`
2. Compile: `javac -d bin src/*.java src/model/*.java src/ui/*.java`
3. Run: `java -cp bin Main` (GUI) or uncomment console version in `Main.java`.

In the GUI:

- Click "Start Hacking Race" to begin.
- Observe progress; restart as needed.

5 Observations and Learnings

Running the simulation multiple times shows variability due to thread scheduling:

- Ascending/Descending hackers perform linearly (O(n) time), often succeeding if the password is near their starting point.
- Random hacker adds unpredictability but avoids repeats.
- Binary search (if enabled) is far more efficient (O(log n)), often winning quickly.
- The 10-second police timer ensures termination, highlighting time-bound concurrency.

This exercise reinforced understanding of Java threads, including creation (`extend Thread`), starting, priorities, and sleeps. It also illustrated challenges like non-deterministic outcomes and the need for clean termination.

6 Answers to Discussion Questions

The assignment includes the following questions for discussion. Answers are provided based on the base implementation (ascending and descending hackers) unless otherwise noted.

1. **What is the probability of winning for each of the hackers or the police?**
   Assuming the password is uniformly random between 0 and 9999, and threads run concurrently without significant overhead, each guess takes exactly 5 ms, and the police timer is 10 seconds (10,000 ms). A hacker can make up to 2,000 guesses in 10 seconds (10,000 ms / 5 ms).
   The ascending hacker wins if the password $\leq$ 1999 (probability 0.2).
   The descending hacker wins if the password $\geq$ 8000 (probability 0.2).
   The police wins otherwise (probability 0.6).
   These probabilities assume ideal parallel execution on multi-core systems; actual results may vary due to OS scheduling.

2. **Why did we use Thread.sleep(5) in the isCorrectPassword function?**
   The `Thread.sleep(5)` simulates the computational time required for each password verification attempt, making the race more realistic by introducing a delay that mimics real-world processing. Without it, the loops would complete almost instantly, trivializing the concurrency aspect.

3. **What will happen if the password is 5000? Who will win?**
   For password 5000:
   - Ascending hacker: 5001 guesses $\times$ 5 ms = 25,005 ms.
   - Descending hacker: 5000 guesses $\times$ 5 ms = 25,000 ms.
   Both is greater thatn 10,000 ms police timer.
   Thus, the police timer expires first, and the police wins.

4. **How can we improve the hackers' chances of winning?**
   - Add more hacker threads with different strategies, such as random guessing (implemented) or binary search (optional in this project), to cover the search space faster.
   - Reduce the verification delay (though this alters the simulation).
   - Use thread pools or optimize guessing to skip duplicates across threads.
   - In this implementation, enabling the binary search hacker drastically improves chances, as it finds any password in at most $\lceil \log_2(10000) \rceil \approx 14$ guesses ($\sim$70 ms).

5. **What are the security risks of using System.exit() from different threads?**
   Calling `System.exit(0)` abruptly terminates the JVM, potentially leaving resources (e.g., files, sockets) uncleaned, leading to data corruption or leaks. In a multi-threaded environment, it allows any thread to shut down the application, which could be exploited if threads are untrusted (e.g., in a server). It prevents graceful shutdown, ignoring cleanup code like finally blocks or shutdown hooks.

6. **Why did we use Thread.MAX_PRIORITY for the hackers? Does this guarantee their win?**
   `Thread.MAX_PRIORITY` gives hackers higher scheduling priority over the police thread, increasing their CPU time allocation to improve chances of cracking the password before the timer. However, priorities are OS-dependent hints, not guarantees— the JVM or OS may ignore them, and factors like system load or single-core execution can still allow the police to win.

7 Conclusion

The "Vault Hacking Race" successfully implements the assignment as a hands-on introduction to multi-threading. By simulating a competitive scenario, it makes abstract concepts tangible. Future improvements could include explicit synchronization for more complex interactions or integrating Java's ExecutorService for better thread management.