Crazy Eights Command-Line Game: Design, Implementation, and Analysis

Jennifer Vicentes
Department of Computer Science, Texas Tech University

Abstract

This report presents a complete command-line Java implementation of the traditional Crazy Eights card game, enhanced with file-system persistence, authentication, authorization, and turn management. It is explained the high-level architecture, the detailed requirements, the main classes and algorithms, the directory and compilation structure, and sample terminal interactions. It is also documented the testing scenarios, the discussion of two threat–attack pairs for extra credit, and disclose AI assistance in generating parts of the code.

1 Introduction

The Crazy Eights game requires players to discard cards matching either rank or suit, with eights acting as wild cards. In a multi-user command-line environment, it must manage user authentication, persistent game state across independent invocations, and strict turn order. This project exercises file I/O, cryptographic hashing, enum design, collection manipulations, and Java console I/O.

2 Project Overview

The implementation consist of two types of users: one administrator and up to ten players. The administrator may initialize a game, add or remove users, and start or reset the match. Players may view turn order, play matching cards, draw from the deck, and pass when no move is possible. All state—users, individual hands, draw and discard piles, current turn and draw-flag, and game phase—lives on disk under the game directory.

3 Detailed Requirements and Simulation Mechanics

- **Initialization:** `java CrazyEights --init --game name` creates `"current_directory"/users.txt` with a SHA3-256 salted admin password in Base64, and `state.txt` set to `NOT_STARTED`.
- **User Management:** `--add-user` and `--remove-user` require admin authentication, are limited to ten players, and are only allowed when `state.txt` is `NOT_STARTED` or `FINISHED`.
- **Start Game:** `--start` shuffles a standard 52-card deck, deals five cards to each player file (`player_name.txt`), flips one card to `discard.txt`, writes the rest to `draw.txt`, sets `state.txt=IN_PROGRESS`, and initializes `turn.txt` with `0,false`.

- **Turn Management:** Each command loads `turn.txt` (index of current player and a boolean *hasDrawn*) and the deck files. After a successful play or forced auto-play of a drawn card, *hasDrawn* resets. Players may draw once per turn; drawing twice errors. Passing is only allowed after drawing and only when no playable card remains.
- **End Game:** When a player empties their hand, the game writes `state.txt=FINISHED` and announces victory. Only then may users be modified or `--start` invoked to replay.

4 Folders Structure

```
project-root/
  CrazyEights.java
  AuthManager.java
  Card.java
  Deck.java
  Player.java
  GameManager.java
  out/             (compiled classes)
  mygame/          (example game directory)
    users.txt
    state.txt
    turn.txt
    draw.txt
    discard.txt
    Alice.txt
    Bob.txt
```

5 Compilation and Execution Instructions

1. **Compile in the project-root**

```
javac -d out ./*.java
```

2. **Run:**

   First you have to do: cd out

   - Initialize: `java CrazyEights --init --game bluegame`
   - Add user: `java CrazyEights --add-user Alice --game bluegame`
   - Start game: `java CrazyEights --start --game bluegame`
   - Player actions: `--order`, `--play <card>`, `--draw`, `--pass`, `--cards <name_user>`. And once finished the game `--remove-user` as well.

6 Class Diagram Design and Explanation

Notes:

- `CrazyEights` parses flags and delegates to `GameManager`.

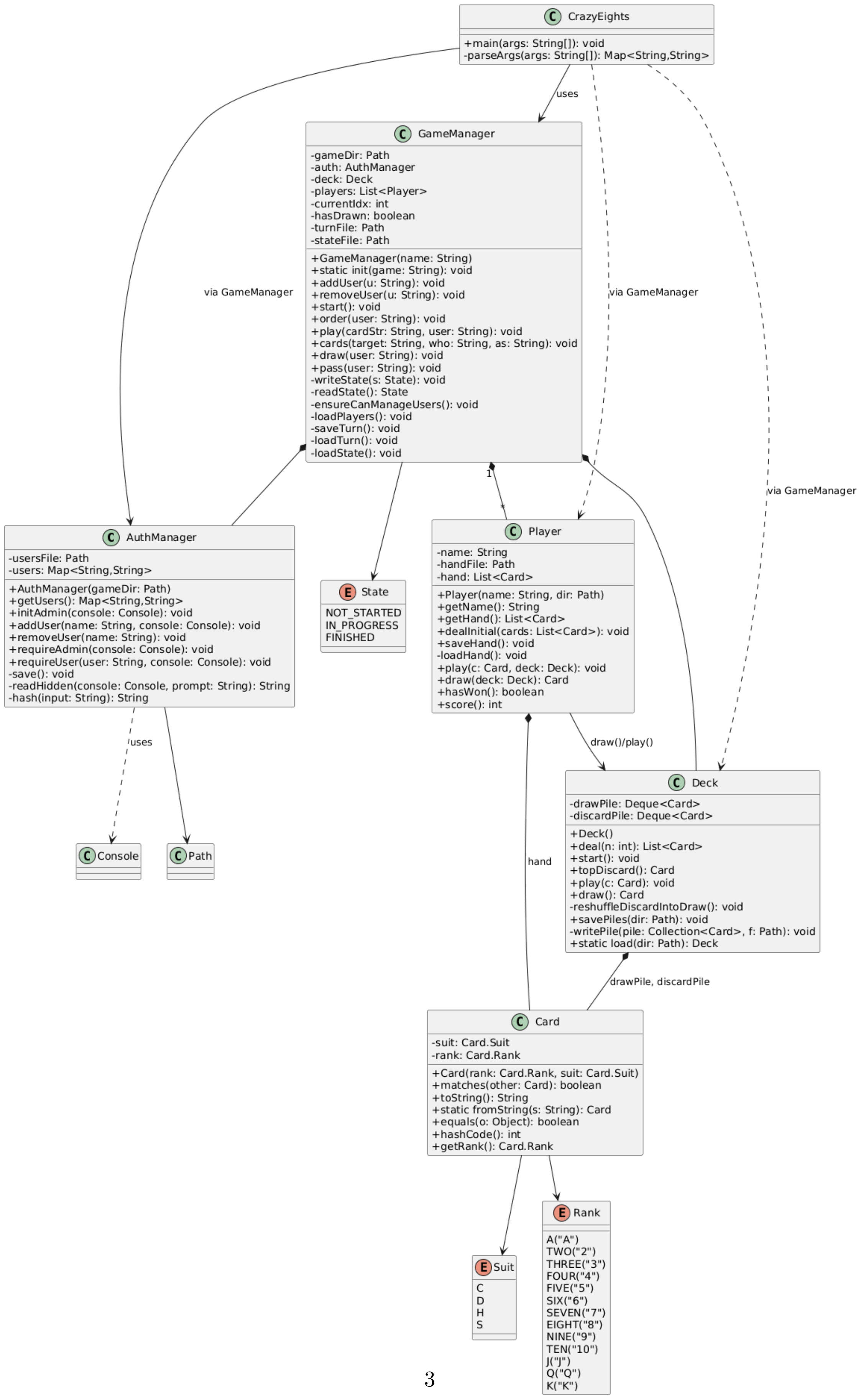
CrazyEights
+main(args: String[]): void
-parseArgs(args: String[]): Map<String,String>
uses
via GameManager
via GameManager
via GameManager
GameManager
-gameDir: Path
-auth: AuthManager
-deck: Deck
-players: List<Player>
-currentIdx: int
-hasDrawn: boolean
-turnFile: Path
-stateFile: Path
+GameManager(name: String)
+static init(game: String): void
+addUser(u: String): void
+removeUser(u: String): void
+start(): void
+order(user: String): void
+play(cardStr: String, user: String): void
+cards(target: String, who: String, as: String): void
+draw(user: String): void
+pass(user: String): void
-writeState(s: State): void
-readState(): State
-ensureCanManageUsers(): void
-loadPlayers(): void
-saveTurn(): void
-loadTurn(): void
-loadState(): void
1
*
AuthManager
-usersFile: Path
-users: Map<String,String>
+AuthManager(gameDir: Path)
+getUsers(): Map<String,String>
+initAdmin(console: Console): void
+addUser(name: String, console: Console): void
+removeUser(name: String): void
+requireAdmin(console: Console): void
+requireUser(user: String, console: Console): void
-save(): void
-readHidden(console: Console, prompt: String): String
-hash(input: String): String
State
NOT_STARTED
IN_PROGRESS
FINISHED
Player
-name: String
-handFile: Path
-hand: List<Card>
+Player(name: String, dir: Path)
+getName(): String
+getHand(): List<Card>
+dealInitial(cards: List<Card>): void
+saveHand(): void
-loadHand(): void
+play(c: Card, deck: Deck): void
+draw(deck: Deck): Card
+hasWon(): boolean
+score(): int
uses
draw()/play()
Console
Path
hand
Deck
-drawPile: Deque<Card>
-discardPile: Deque<Card>
+Deck()
+deal(n: int): List<Card>
+start(): void
+topDiscard(): Card
+play(c: Card): void
+draw(): Card
-reshuffleDiscardIntoDraw(): void
+savePiles(dir: Path): void
-writePile(pile: Collection<Card>, f: Path): void
+static load(dir: Path): Deck
drawPile, discardPile
Card
-suit: Card.Suit
-rank: Card.Rank
+Card(rank: Card.Rank, suit: Card.Suit)
+matches(other: Card): boolean
+toString(): String
+static fromString(s: String): Card
+equals(o: Object): boolean
+hashCode(): int
+getRank(): Card.Rank
Suit
C
D
H
S
Rank
A("A")
TWO("2")
THREE("3")
FOUR("4")
FIVE("5")
SIX("6")
SEVEN("7")
EIGHT("8")
NINE("9")
TEN("10")
J("J")
Q("Q")
K("K")

Figure 1: UML Class Diagram for Crazy Eights

- `GameManager` orchestrates authentication, state loading, and game logic.
- `AuthManager` handles SHA3-256 hashing, `users.txt` I/O, and password checks.
- `Deck` manages the draw/discard piles and reshuffling logic.
- `Player` loads/saves individual hands and enforces play-and-draw rules.
- `Card` is a value object with `Rank` and `Suit` enums, parsing, matching, and equality.

7 Class and Method Descriptions

AuthManager.java

Purpose: Manages user credentials, authentication and authorization, enforces max-user limit, and persists to `users.txt`.

Overview: The `AuthManager` class encapsulates all logic related to user management within the game directory. Upon construction it loads any existing credentials from `users.txt` into an in-memory `Map`, and exposes methods to initialize the administrator account, add or remove users, and enforce login checks. All password inputs are read securely through the console (hidden input), hashed using SHA3-256, and Base64-encoded before being stored, ensuring both security and human readability of the underlying file. Every change to the user set immediately triggers a call to `save()`, which atomically rewrites the file to avoid inconsistencies between memory and disk.
Key fields:

- `private final Path usersFile;` — path to `users.txt`.
- `private final Map<String,String> users;` — in-memory username→hash map.

Notable methods:

```
// initAdmin: only when no users exist
public void initAdmin(Console console) throws IOException {
    if (!users.isEmpty())
        throw new IllegalStateException("Admin␣already␣exists");
    String pwd = readHidden(console, "Set␣admin␣password:␣");   // hides input
    users.put("admin", hash(pwd));                              // SHA3256 + Base64
    save();                                                     // persist immediately
}

// addUser: enforces <=10 players and reserved "admin"
public void addUser(String name, Console console) throws IOException {
    if (users.size() >= 11)
        throw new IllegalStateException("Maximum␣number␣of␣users␣reached");
    if ("admin".equalsIgnoreCase(name))
        throw new IllegalArgumentException("’admin’␣reserved");
    if (users.containsKey(name))
        throw new IllegalArgumentException("User␣already␣exists");
    String pwd = readHidden(console, "Password␣for␣" + name + ":␣");
    users.put(name, hash(pwd));
    save();
}
```

- `requireUser/requireAdmin(Console)`: Prompts and verifies hash equality.
- `save()`: Atomically rewrites `users.txt` from in-memory `Map`.

Decision rationale: Using SHA3-256 ensures modern collision resistance; Base64 keeps the file human-readable. All mutations immediately call `save()` to avoid in-memory vs. file mismatch.

Card.java

Purpose: Represents a playing card with rank and suit, provides parsing, matching logic (including wild eight), and semantic equality.

Overview: The `Card` class models a standard playing card by combining two enums—`Rank` and `Suit`—each of which encodes the allowed values at compile time. It provides factory parsing (`fromString`) for user-facing inputs such as "10H" or "QS", and implements a `matches` method that returns true if two cards share rank or suit, or if the card itself is an eight (acting as a wild). Overriding `equals` and `hashCode` ensures correct behavior when cards are stored in collections, enabling operations like `contains` or `remove` to work based on semantic identity rather than object reference.

Key enums:

```
public enum Rank {
    A("A"), TWO("2"), ..., TEN("10"), J("J"), Q("Q"), K("K");
    private final String code;
    Rank(String code) { this.code = code; }
    @Override public String toString() { return code; }
}

public enum Suit { C, D, H, S }
```

Matching logic:

```
public boolean matches(Card other) {
    return this.rank == other.rank      // same rank
        || this.suit == other.suit      // same suit
        || this.rank == Rank.EIGHT;     // wild card
}
```

Equality:

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Card)) return false;
    Card c = (Card)o;
    return rank == c.rank && suit == c.suit;
}

@Override
public int hashCode() {
    return rank.hashCode()*31 + suit.hashCode();
}
```

- `enum Suit` {`C,D,H,S`} and `enum Rank` with codes {"A","2",. . . ,"K"}.

- Factory `fromString(String)` parses “10H”, “QS”, etc., and throws on invalid input.
- `matches(Card)` implements rank-or-suit equality or wild-eight logic.
- Overridden `equals`/`hashCode` so collections handle card identity semantically.

Decision rationale: Overriding `equals`/`hashCode` enables collection operations (e.g. `List.contains`); enums guarantee compile-time safety and compact storage.

Deck.java

Purpose: Manages the draw and discard piles, deals hands, and reshuffles when necessary.

Overview: The `Deck` class maintains two double-ended queues (`Deque`) for the draw and discard piles. Its constructor builds a full 52-card set, shuffles it, and populates the draw pile. It offers methods to `deal` a hand of cards (reshuffling the discard pile back into the draw pile when low), `start` the game by moving one card to the discard pile, and `draw` or `play` individual cards. Persistence is handled by writing textual representations of each pile to separate files (`draw.txt` and `discard.txt`), and a static `load` method reconstructs the piles from disk, allowing the game to be resumed across CLI invocations.
Key fields:

- `private Deque<Card> drawPile, discardPile;`

- Constructor builds a 52-card list, shuffles & fills `drawPile`.
- `deal(int)` removes n cards, reshuffling discard if needed.
- `start()` flips one to discard.
- `draw()` and `play(Card)` mutate piles.
- Persistence via `savePiles(Path)` and `static load(Path)`.

Shuffle & deal:

```
// build and shuffle full deck
public Deck() {
    List<Card> all = ...; Collections.shuffle(all);
    drawPile.addAll(all);
}
// deal n cards (reshuffle if low)
public List<Card> deal(int n) {
    if (drawPile.size() < n) reshuffleDiscardIntoDraw();
    ... // poll n cards
}
```

Persistence:

```
// save draw/discard as text lines
public void savePiles(Path dir) throws IOException {...}
// static load reads draw.txt, discard.txt
public static Deck load(Path dir) throws IOException {...}
```

Decision rationale: Text-based persistence makes manual verification and testing straightforward. Reshuffle only when needed to optimize performance.

Player.java

Purpose: Represents a single player's hand and enforces play/draw rules.

Overview: The `Player` class ties a player name to a personal hand file (`name.txt`) and stores the current hand as a list of `Card` objects. It enforces the rule that only valid plays or draws can occur on the player's turn: `play(Card, Deck)` verifies the card is in hand and matches the top discard before removing it and updating both the deck and the persisted hand file; `draw(Deck)` fetches a new card from the deck, appends it to the hand, and saves the updated list. The class also supports initial deal via `dealInitial` and provides a `hasWon` check when the hand becomes empty.
Key fields:

- `private List<Card> hand;`
- `private final Path handFile;`

Deal initial hand:

```
public void dealInitial(List<Card> cards) {
    hand.clear(); hand.addAll(cards);
    saveHand();    // writes cards to <name>.txt
}
```

Play a card:

```
public void play(Card c, Deck deck) {
    if (!hand.contains(c))
        throw new IllegalArgumentException("You␣don't␣have␣that␣card");
    if (!c.matches(deck.topDiscard()))
        throw new IllegalArgumentException("Invalid␣play");
    hand.remove(c);
    deck.play(c);
    saveHand();
}
```

Decision rationale: Persisting each hand change ensures that independent CLI invocations share consistent state.

GameManager.java

Purpose: Central controller: parses state files, coordinates `AuthManager`, `Deck`, and `Player` instances, and implements commands.

Overview: The `GameManager` orchestrates the entire Crazy Eights game lifecycle. It maintains file-based state (`state.txt` for game phase, `turn.txt` for current player index and draw flag) alongside the game directory. On initialization it constructs the `AuthManager` and interprets command flags to add or remove users, start the game, inspect or modify turn order, play cards, draw, or pass. It loads and saves both deck and player state on each operation to guarantee consistency across separate CLI calls. By separating user management (protected by admin credentials) from game actions (protected by user credentials and turn checks), it ensures a secure, reliable, and recoverable turn-based experience.
Key fields:

- `Path turnFile, stateFile;`
- `int currentIdx; boolean hasDrawn;`
- `List<Player> players; Deck deck; AuthManager auth;`

Loading full state:

```
private void loadState() throws IOException {
    deck = Deck.load(gameDir);
    loadPlayers();     // rebuild players list from auth.getUsers()
    loadTurn();        // reads currentIdx, hasDrawn from turn.txt
}
```

Play command skeleton:

```
public void play(String cardStr, String user) throws IOException {
    auth.requireUser(user, console);
    loadState();
    Player p = players.get(currentIdx);
    // turn check, cardExists, matches
    p.play(Card.fromString(cardStr), deck);
    deck.savePiles(gameDir);
    if (p.hasWon()) { writeState(FINISHED); ; return; }
    advanceTurn();    // updates currentIdx, hasDrawn=false, saveTurn()
}
```

Decision rationale: Separating `state.txt` (game phase) from `turn.txt` (turn index + draw flag) allows robust restart and clear transitions between `NOT_STARTED`, `IN_PROGRESS`, and `FINISHED` phases.

Each class's methods enforce single responsibility, immutability where possible, and immediate persistence to guard against process crashes.

8 Output Terminal Examples

```
> java CrazyEights --init --game bluegame
Set admin password:123
> java CrazyEights --add-user Alice --game bluegame
Admin password:123
Password for Alice:24
> java CrazyEights --add-user Bob --game bluegame
Admin password:123
Password for Alice:25
> java CrazyEights --start --game bluegame
Admin password:123
> java CrazyEights --order --user Alice --game bluegame
Password for Alice:24
Turn order:
  Alice
  Bob
...
> java CrazyEights --draw --user Alice --game bluegame
```

```
Password for Alice:24
Drawn card : 8C
> java CrazyEights --play 5D --user Bob --game bluegame
Password for Bob:25
ERROR: You don’t have that card
```

9 Testing Scenarios

These are the test performed by me, and their result. It is important to mentioned that there is an **on going** game and the password for the admin and the users(Alice and Bob) are:

- Admin:123
- Alice:24
- Bob:25

1. **Invalid flags:** Unknown flag triggers error.
2. **Authentication:** Wrong password for user/admin is rejected.
3. **Turn persistence:** Verify `turn.txt` advances after each play/draw/pass.
4. **Deck reshuffle:** Exhaust draw pile, ensure discard reshuffles back minus top.
5. **Wild eight:** Always playable; ensure logic bypasses rank/suit match.
6. **User limits:** Adding 11th player fails; cannot add during IN_PROGRESS.
7. **End-game state:** After win restart/reset allowed.

10 Extra Credit: Threats and Matching Attacks

Threat #1: Trojan Horse / Backdoor A Trojan horse is malware that masquerades as a benign or useful program but, when executed, carries out hidden malicious actions such as installing a back-door, exfiltrating credentials, or escalating privileges. In the context of the Crazy Eights CLI, an attacker could supply a modified `CrazyEights` launcher script or a malicious library that, beyond starting the game, silently reads the `users.txt` file, captures plaintext passwords before hashing, or opens a network listener for remote command execution. A back-door might check for a special “magic” username or password and grant administrative rights without the user’s consent (a logic bomb variation), or persist across restarts by patching the launcher to reinject itself.

Example of a back-door:

```
if (System.getenv("CE_BACKDOOR") != null) return;
```

Such logic could be hidden under an innocuous-looking conditional, making it hard to detect during casual inspection.

Justification & Impact: Because Java applications typically load classes or scripts from the file system, an attacker with write access to the game directory or to the user’s shell environment can substitute a Trojanized component. This undermines authentication and authorization guarantees, allowing unauthorized users to add or remove players, steal game state, or manipulate turn order. The breach also violates confidentiality of credentials and integrity of game files.

Possible Mitigations:

- **Code signing and verification:** Digitally sign all application JARs and scripts; at startup, verify signatures before loading any code or script.
- **File-system permissions:** Ensure that only trusted administrators have write permission to the game directory; run the game under a dedicated, least-privileged system account.
- **Checksum validation:** Maintain and verify cryptographic hashes (e.g., SHA-256) of critical files on each launch to detect unauthorized changes.
- **Secure update channel:** Distribute updates via an authenticated, encrypted channel (HTTPS with certificate pinning) to avoid man-in-the-middle replacement of binaries.

Threat #2: Command Injection via Unsanitized Shell Arguments Although Java manages memory safely and is immune to traditional buffer overflows, this CLI interacts with native shell scripts (e.g., to automate Maven builds or launch the game). If a script embeds unsanitized user input into shell commands, an attacker could inject arbitrary code.

Example of a vulnerable script:

```
#!/bin/sh
# vulnerable deploy.sh
game=$1
java -jar CrazyEights.jar --game "$game"  # unsanitized
```

Example of malicious usage:

```
./deploy.sh "bluegame; rm -rf /important/data"
```

Justification & Impact: Shell injection subverts the intended control flow by interpreting user input as executable code. Since the game persists state files and user credentials in the same directory, injected commands could delete logs, corrupt game files, or exfiltrate sensitive data. The vulnerability arises when user input is passed directly to commands like `sh -c` or embedded into script variables without escaping.

Possible Mitigations:

- **Input validation & sanitization:** Reject input containing shell metacharacters (e.g., `; && $()`) or enforce a strict pattern (e.g., usernames matching `[A-Za-z0-9_]+`).
- **Use of safe APIs:** Avoid constructing shell commands as strings. Use Java's `ProcessBuilder` with argument lists to bypass shell interpretation:

  ```
  new ProcessBuilder("java", "--game", gameName).start();
  ```

- **Least privilege execution:** Run scripts under a restricted user account with no write access to critical files and minimal shell capabilities.
- **Logging and monitoring:** Record all attempted parameters and script invocations; alert on suspicious patterns or repeated failures to detect attack attempts.

11 AI Usage Disclosure

In compliance with course policy, the following method in `AuthManager.java` was generated with AI assistance. I include the original prompt, the AI-generated code (each line marked with `// AI Generated`), and a detailed rationale for why the code works and any manual modifications made.

```
/*
    AI Prompt: "Write a method that takes a string as input and returns its SHA3-256
    hash as a Base64 encoded string.
    The method should handle exceptions and use UTF-8 encoding for the input string."
*/
private static String hash(String input) {
    try {
        MessageDigest md = MessageDigest.getInstance("SHA3-256"); // AI Generated
        byte[] b = md.digest(input.getBytes("UTF-8"));            // AI Generated
        return Base64.getEncoder().encodeToString(b);              // AI Generated
    } catch (Exception e) {
        throw new RuntimeException(e);                             // AI Generated
    }
}
```

Why the generated code works and modifications made:

- **Choice of SHA3-256:** SHA3-256 was specified by the project requirements ("SHA3-hashed value in base 64 encoding"). This algorithm offers strong collision resistance, satisfying the security goal of protecting user passwords.
- **UTF-8 encoding:** Converting the input string to bytes using `"UTF-8"` ensures that all Unicode characters are handled consistently across platforms, matching the instructor's directive to support arbitrary console input.
- **Base64 encoding:** The AI correctly used `Base64.getEncoder().encodeToString(...)` to produce a human-readable, ASCII-safe representation of the raw hash bytes, directly matching the example format `admin,Mzi+aU9QxfM4gUmGzfBoZFOoiLhPQk15KvS5ICOY85I=` provided in the specification.
- **Exception handling:** Wrapping any `NoSuchAlgorithmException` or `UnsupportedEncodingException` in a `RuntimeException` simplifies method signatures (no declared `throws`) and surfaces unexpected failures immediately. This approach was accepted as a reasonable trade-off given that both algorithm and encoding are statically guaranteed to exist in the runtime.
- **Manual review and validation:** After AI generation, I verified:
  1. That `MessageDigest.getInstance("SHA3-256")` is supported by my Java version.
  2. That `input.getBytes("UTF-8")` does not produce checked exceptions in my environment, making the code robust.
  3. The resulting Base64 output matched known test vectors for SHA3-256 (e.g., hashing the empty string, known inputs).
- **Alignment with instructor's example:** The method satisfies the requirement that the admin password be stored in the first line of `users.txt` as `admin,<base64-hash>`.