Project Report: Decentralized Secure Event Ticketing Platform

Date: December 15, 2025 **Author:** Cihan Aygün - 211401010

---

1. Introduction

This project implements a **Decentralized Event Ticketing Platform (DApp)** that addresses common issues in the traditional ticketing industry, such as scalping, lack of transparency, and inefficient refund processes. By leveraging **blockchain technology**, specifically the Ethereum network (simulated via Hardhat), we create a secure, transparent, and user-centric ticketing ecosystem.

Key features include:

- **NFT-Based Tickets:** Each ticket is a unique ERC-721 token.
- **Transparent Resale:** Refunded tickets are returned to the pool for resale at face value, preventing price manipulation.
- **Organizer Controls:** Event creators can scan tickets and manage event lifecycles.
- **Community Voting:** Attendees can vote on completed events, creating a reputation system for organizers.

---

2. System Architecture

The application follows a modern DApp architecture:

- **Smart Contracts (Backend):** Written in **Solidity**, deployed on a local Hardhat Network.
- **Frontend:** Built with **Next.js (React)**, using **Tailwind CSS** for styling.
- **Blockchain Interaction: Wagmi** and **Viem** hooks are used to connect the frontend to the smart contracts.
- **Wallet Integration: RainbowKit** handles wallet connections (e.g., MetaMask).

Technological Stack

- **Solidity 0.8.20:** Safe and efficient smart contract language.
- **OpenZeppelin:** Standard libraries for ERC-721 and Ownable patterns.
- **Hardhat:** Development environment for compiling, testing, and deploying contracts.
- **TypeScript:** Ensures type safety across the full stack.

---

3. Smart Contract Implementation

The core logic is divided into two main contracts: `TicketingFactory` and `TicketingContract`.

3.1. Factory Pattern (`TicketingFactory.sol`)

To allow any user to create an event, we use a Factory pattern. This contract deploys a new instance of `TicketingContract` for each event, ensuring isolation and scalability.

Code Snippet: Creating an Event

```
function createEvent(
    string memory _name,
    string memory _description,
    uint256 _maxSupply,
    uint256 _ticketPrice,
    uint256 _eventStartDate,
    uint256 _entryDuration
) external returns (address) {
    // Deploy new contract
    TicketingContract newEvent = new TicketingContract(
        msg.sender, // Initial Owner
        _name,
        _description,
        _maxSupply,
        _ticketPrice,
        _eventStartDate,
        _entryDuration
    );

    address eventAddr = address(newEvent);
    allEvents.push(eventAddr);
    organizerEvents[msg.sender].push(eventAddr);

    emit EventCreated(eventAddr, msg.sender, _name);

    return eventAddr;
}
```

3.2. Ticket Logic (`TicketingContract.sol`)

This contract manages the lifecycle of tickets for a specific event.

Minting & Resale Logic

A critical innovation in this project is the **Refund/Resale Mechanism**. Instead of permanently burning refunded tickets, they are returned to the contract. When a new user tries to buy a ticket, the contract prioritizes selling these "refunded" tickets first.

Code Snippet: Minting with Priority Resale

```
function mintTicket() external payable nonReentrant returns (uint256 tokenId) {
    if (msg.value != TICKET_PRICE) revert("Incorrect Value");

    // Check Resale Queue (Refunded Tickets)
    if (refundedTicketIds.length > 0) {
        // Pop from stack
        tokenId = refundedTicketIds[refundedTicketIds.length - 1];
        refundedTicketIds.pop();

        // Critical Fix: Use _transfer to bypass approval checks since Contract is owner
        _transfer(address(this), msg.sender, tokenId);

        tickets[tokenId].status = STATUS_ACTIVE;
        stats.totalSold++;
    } else {
        // Standard Minting
        if (stats.totalMinted >= MAX_SUPPLY) revert SupplyExhausted();
        stats.totalMinted++;
        tokenId = stats.totalMinted;

        _safeMint(msg.sender, tokenId);
        tickets[tokenId].status = STATUS_ACTIVE;
        stats.totalSold++;
    }
}
```

Refund Mechanism

Users can request a refund up to 6 hours before the event. The ticket is transferred back to the contract, and the user is refunded.

Code Snippet: Refund Logic

```
function requestRefund(uint256 tokenId) external nonReentrant {
    if (block.timestamp >= refundDeadline) revert RefundPeriodExpired(block.timestamp, refundDeadline);

    // Transfer ticket to contract
    transferFrom(msg.sender, address(this), tokenId);

    // Update State: Mark as FOR_SALE
    tickets[tokenId].status = STATUS_FOR_SALE;
    refundedTicketIds.push(tokenId);
    stats.totalSold--;

    // Refund Money
    payable(msg.sender).transfer(TICKET_PRICE);
}
```

Voting Mechanism

Users can request a refund up to 6 hours before the event. The ticket is transferred back to the contract, and the user is refunded.

Code Snippet: Voting Logic

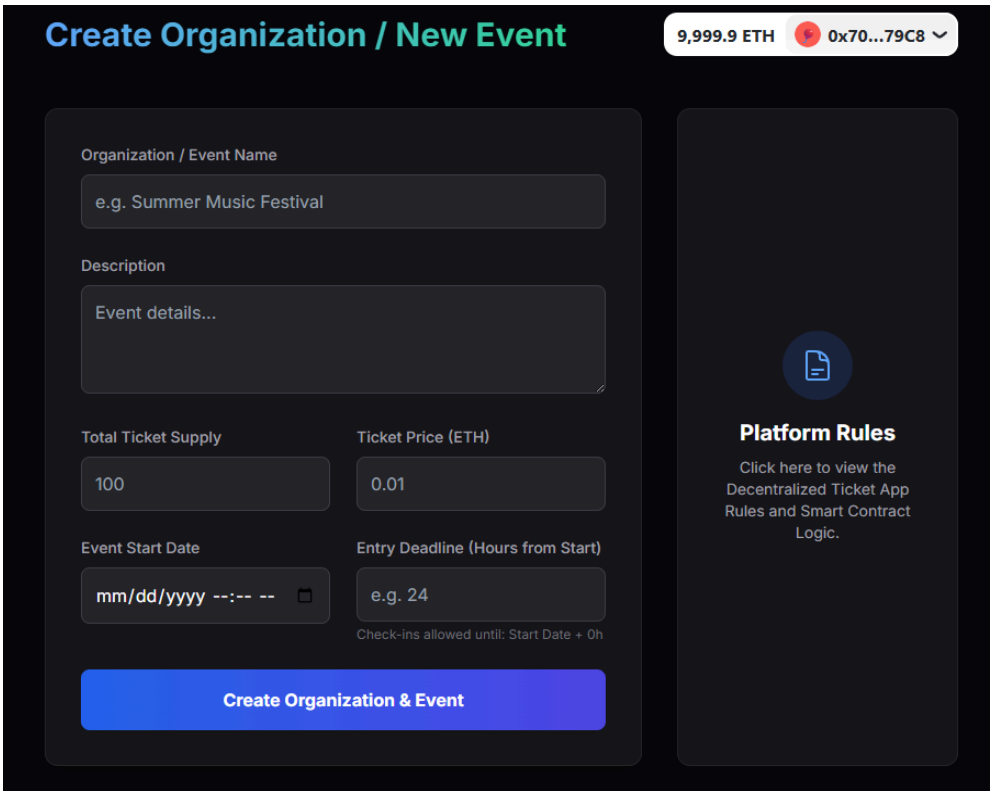
```
function vote(uint256 tokenId, bool isPositive) external {
     if (block.timestamp > votingDeadline) revert VotingPeriodExpired(block.timestamp, votingDeadline);
     // Can only vote if they attended (scanned -> BURNED)
     if (tickets[tokenId].status != STATUS_BURNED) revert VoteEligibilityFailed(tokenId);
     if (tickets[tokenId].hasVoted) revert AlreadyVoted(tokenId);

     tickets[tokenId].hasVoted = true;
     stats.totalVoted++;
     if (isPositive) {
         stats.positiveVotes++;
     }
 }
```

4. Frontend Implementation

The frontend provides a user-friendly interface for interacting with the blockchain.

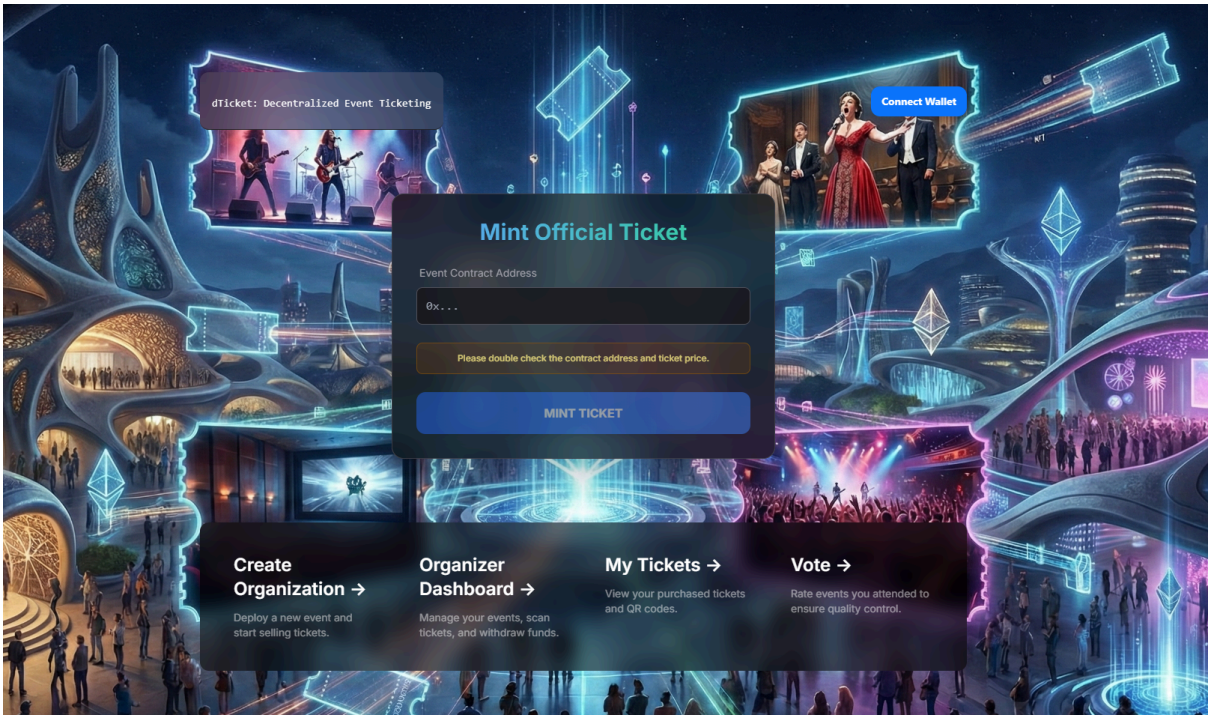
4.1. Creating an Organization

Users can easily deploy their own event contracts by filling out a form.



4.2. Ticket Purchase

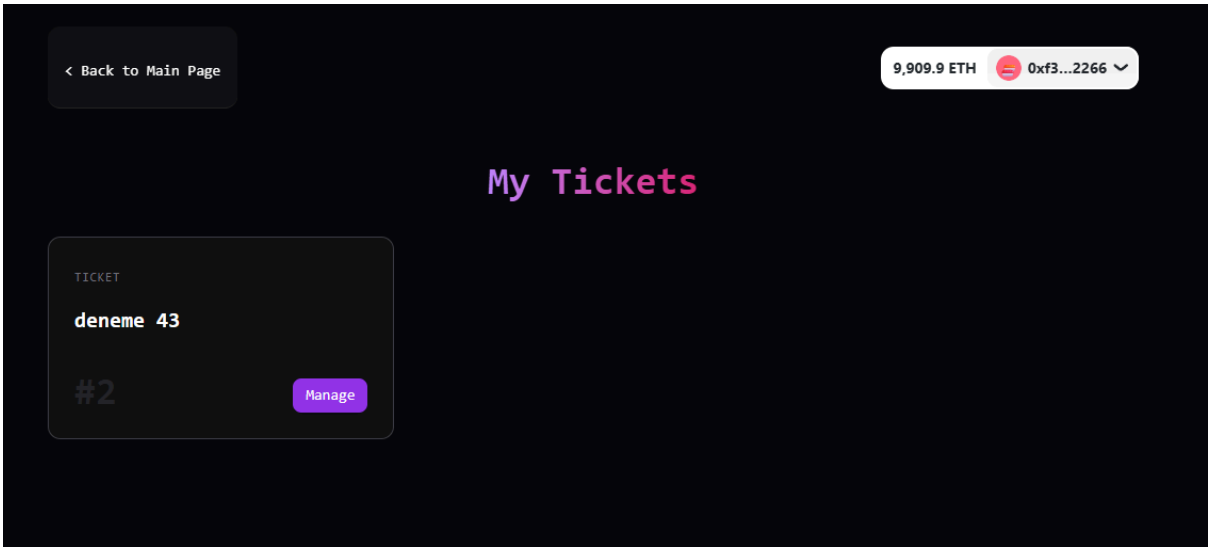
The main landing page allows users to find events and mint tickets. The UI handles wallet connection and transaction feedback.

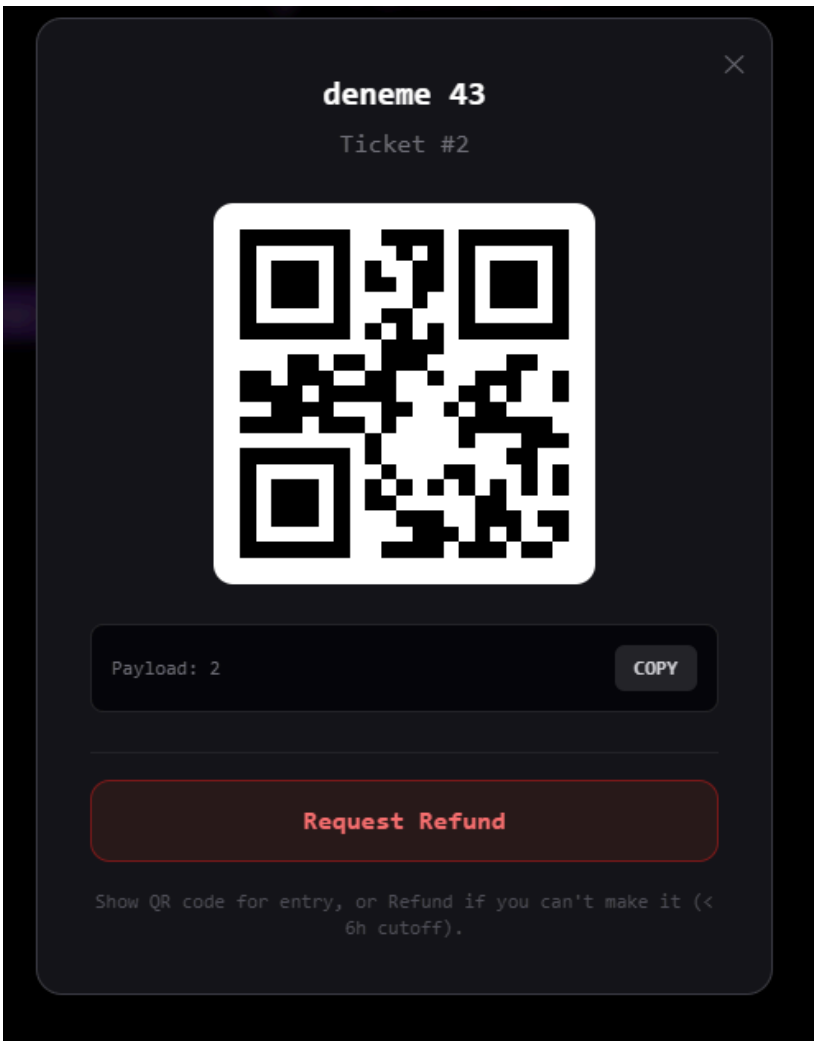


4.3. My Tickets & Refunds

The "My Tickets" page displays the user's tickets. It allows users to:

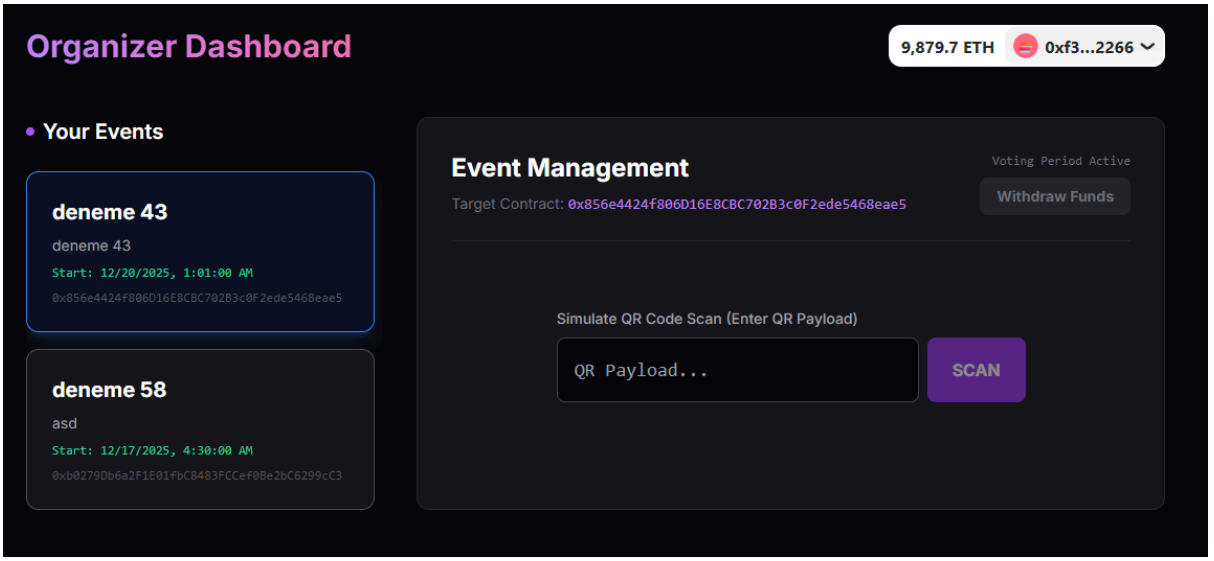
1. **Reveal QR Code:** Generates a cryptographic signature (off-chain) for entry validation.
2. **Request Refund:** Returns the ticket to the pool.





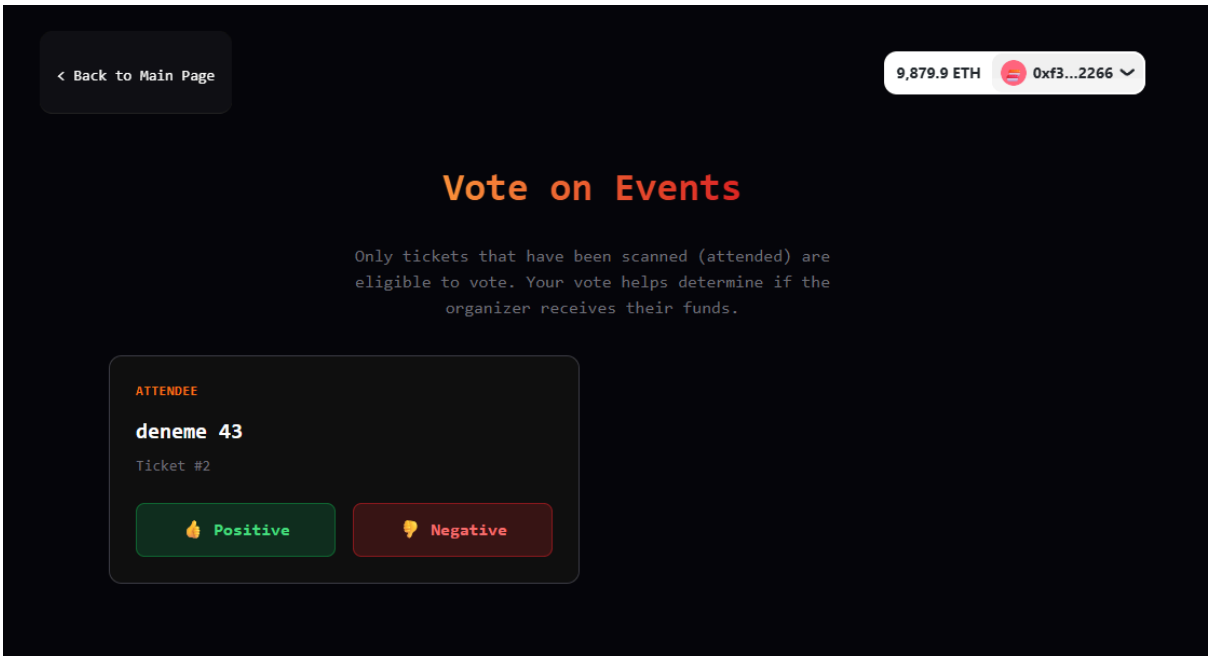
4.4. Organizer Dashboard

Organizers can view their events and use the **Gatekeeper** feature to scan QR codes. This validates attendance on-chain.



4.5. Voting Page

After an event, attendees with scanned tickets can access the Voting Page to cast a positive or negative vote. This feedback helps future attendees evaluate organizer quality.



5. Challenges and Solutions

Challenge: Reselling Refunded Tickets

Problem: Initially, when a refunded ticket (owned by the contract) was being sold to a new user, the transaction failed with an "Unauthorized" error. This happened because the standard `transferFrom` function requires the sender (`msg.sender`) to be approved, but in this case, the *contract itself* is the owner and initiator.

Solution: We modified the smart contract to use the internal `_transfer` function (from OpenZeppelin's `ERC721`) instead of `transferFrom` within the `mintTicket` function. This allows the contract to securely transfer its own tokens to the buyer without needing an external approval transaction.

Challenge: Frontend Configuration

Problem: Every time the contracts were redeployed, the contract address changed, breaking the frontend until manually updated.

Solution: We implemented an automated deployment script. Now, `deploy.ts` writes the new contract address directly to a shared TypeScript configuration file (`frontend/utils/contractAddress.ts`), ensuring the frontend is always in sync.

6. Conclusion

This project successfully demonstrates a functional decentralized ticketing system. It solves key industry problems through transparent resale mechanisms and secure ownership verification. The combination of a robust Solidity backend and a reactive Next.js frontend provides a seamless user experience comparable to Web2 applications.