Technical Assessment for Betfin Project

Section 1: Practical Coding Challenge

1. Smart Contract Development:
   - **Task:**
     Write a simple ERC-721 smart contract for an NFT collection. The contract should include functionality for minting, transferring, and querying ownership. Ensure to implement basic security best practices.
   - **Requirements:**
     - Display wallet address of user and metadata of NFT that user owns when the user login with Web3 wallet from the profile page of Betfin.
     - Include a function that allows users to mint a new NFT with metadata (name, description, rarity).

2. Code Review:

- **Task:** Analyze the following code snippet for vulnerabilities and inefficiencies. Suggest improvements and explain your rationale.

```
function transfer(address _to, uint256 _tokenId) public {
    require(ownerOf(_tokenId) == msg.sender, "Not the owner");
    owners[_tokenId] = _to;
}
```

The provided function allows a token owner to transfer ownership of a token to another address. However, it has several **security vulnerabilities** and **logical inefficiencies** that could lead to inconsistent state or loss of tokens.

Issues Identified

1. **No validation for zero address**
   The function doesn't prevent transferring tokens to `address(0)`, which could effectively burn the token unintentionally.

   ```
   require(_to != address(0), "Invalid recipient address");
   ```

2. **No event emission**
   In ERC-721 and ERC-20 standards, every transfer must emit a Transfer event to allow dApps and explorers to track movements.
   Missing this event breaks transparency and interoperability.

3. **No balance tracking**
   The function only updates the owners mapping but doesn't update the sender's and receiver's balances, leading to inconsistent token ownership data.

4. **Lack of access control / approval logic**
   It only checks that the sender is the owner. In ERC-721 standards, approved operators (via approve or setApprovalForAll) should also be allowed to transfer.

5. **Non-standard naming**
   Using transfer() is misleading. ERC-721 and ERC-20 standards use transferFrom() and safeTransferFrom() to ensure compatibility.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

mapping(uint256 => address) private _owners;
mapping(address => uint256) private _balances;
event Transfer(address indexed from, address indexed to, uint256
indexed tokenId);

function transferFrom(address from, address to, uint256 tokenId)
public {
    require(ownerOf(tokenId) == msg.sender, "Caller is not the
owner");
    require(to != address(0), "Cannot transfer to zero address");

    // Update balances
    _balances[from] -= 1;
    _balances[to] += 1;

    // Update ownership
    _owners[tokenId] = to;

    // Emit standard event
    emit Transfer(from, to, tokenId);
}
```

Rationale

- Prevents accidental burns by validating _to.
- Emits Transfer event for transparency and standard compliance.
- Updates _balances to maintain consistent state.
- Uses the standard ERC-721 pattern (transferFrom) for interoperability with wallets and marketplaces.

3. Deployment:

- Provide a brief guide on how to deploy your smart contract to a test network (e.g., Rinkeby or Kovan). Include any necessary configurations and commands.

Deployment Guide: Why Rinkeby and Kovan Should Be Avoided — and How We Use Sepolia Instead

In the past, Ethereum testnets such as **Rinkeby**, **Kovan**, and **Ropsten** were widely used to test and deploy smart contracts. However, since Ethereum transitioned to Proof of Stake (The Merge), **these testnets have been deprecated and are no longer supported by the Ethereum Foundation**.

As of today:

- They are no longer actively maintained.
- Most infrastructure providers like **Infura**, **Alchemy**, and **Etherscan** have dropped support.
- Faucets are unreliable or offline.
- Block explorers are not consistent or even reachable in some cases.

For those reasons, deploying to Rinkeby or Kovan is no longer recommended.

Our smart contract is already deployed on **Sepolia**, the currently recommended testnet by the Ethereum community and Ethereum Foundation. All QA, integration testing, and frontend interactions are running on Sepolia.

Step-by-Step Guide to Deploying on Sepolia

1. Set your environment variables (`.env`)

```
SEPOLIA_RPC_URL=https://sepolia.infura.io/v3/INFURA_KEY
PRIVATE_KEY=0x... (your test wallet's private key with Sepolia ETH)
```

2. Configure `hardhat.config.js`

```
require("@nomiclabs/hardhat-ethers");
require("dotenv").config();
module.exports = {
  defaultNetwork: "sepolia",
  networks: {
    sepolia: {
      url: process.env.SEPOLIA_RPC_URL,
      accounts: [process.env.PRIVATE_KEY],
    },
  },
  solidity: "0.8.20",
};
```

3. Create the deploy script (`scripts/deploy.js`)

```
async function main() {
  const Contract = await ethers.getContractFactory("YourContractName");
  const contract = await Contract.deploy();
  await contract.deployed();
  console.log(`✅ Contract deployed to: ${contract.address}`);
}

main().catch((error) => {
  console.error(error);
  process.exitCode = 1;
});
```

4. Run the deployment

```
npx hardhat run scripts/deploy.js --network sepolia
```

5. Fund your wallet with Sepolia ETH

Use any of the following faucets:

- https://sepoliafaucet.com/
- https://faucet.quicknode.com/ethereum/sepolia

- **Final Notes**
- Sepolia is stable, EVM-compatible, and mirrors mainnet conditions.
- It is supported by modern tooling including **Etherscan**, **Infura**, **Chainlink**, and **OpenZeppelin Defender**.
- Using Sepolia ensures a future-proof testing environment aligned with the current Ethereum roadmap.

4. Bug **Fix:**

- When the user login with Web3 wallet from the Betfin dashboard, display wallet address to user profile.

- When the page reloaded, fix the error that wallet logout automatically and add logout function to user profile.

OK in code

Section 2: **Problem-Solving and Scenario-Based Questions**

1. Scenario:
    - **Task:** Your team is facing performance issues with the current smart contracts due to high gas fees during minting. How would you address this problem? What optimizations would you consider?

High gas fees during minting are a common bottleneck, especially as user adoption grows. To address this, I would take a multi-layered approach:

1. **Optimize Storage Usage:**
   Storage writes are the most expensive operations on-chain. I'd review the minting logic to eliminate redundant storage updates. For example, minimizing the use of SSTORE by packing variables (uint8, bool, etc.) and avoiding unnecessary mappings.
2. **Use ERC721A Instead of ERC721:**
   If we are minting multiple tokens per transaction (e.g., batch minting), I'd suggest switching to ERC721A by Chiru Labs. It significantly reduces gas costs by optimizing how ownership is recorded during batch mints.
3. **Lazy Minting or Off-Chain Metadata:**
   We could consider **lazy minting**, where the token metadata is pre-generated off-chain and only minted on-chain when claimed. This approach defers and reduces gas usage at the moment of mint.
4. **Compress Events & Avoid Unused Emit Logs:**
   Events are useful for indexing, but emitting unnecessary logs can be costly. I'd clean up the events emitted during minting to include only essential data.
5. **Use Immutable Variables and Constants:**
   If the contract uses configuration values (e.g., max supply, base URI), setting them as immutable or constant reduces gas when referenced during execution.
6. **External Call Minimization:**
   Ensure that the mint function doesn't trigger external contract calls (e.g., royalty registry or external validation) unless strictly necessary.

7. **Gas Reporting Tools:**
   I’d also run **Hardhat Gas Reporter** or **Tenderly Trace** to get precise insights into the heaviest parts of the mint function and refactor accordingly.

Lastly, if scalability becomes critical, I’d explore **L2 deployment** options like **Base**, **Arbitrum**, or **Polygon**, where gas fees are significantly lower while keeping EVM compatibility.

2. Real-**World Example:**

- **Question:** Discuss a challenging blockchain project you have worked on. What were the main hurdles, and how did you resolve them?

One tough challenge I faced was keeping the wallet session active after a page refresh. We were using MetaMask with wagmi and ethers, but constant updates in these libraries caused serious compatibility issues.

The biggest problem was that after refreshing, the user had to reconnect manually every time — which ruined the UX.

To fix it, I migrated everything to wagmi v2 with viem, handled session persistence through localStorage, and set up proper mocking for wallet hooks during testing. It was a mix of refactoring, version control, and improving the app’s state management.

In the end, we got a seamless connect experience that survived page reloads and worked across environments.

Deliverables

- Integrate all source code with smart contract code and deployment script to our project repo.
- Short video or screenshot with minting process and live demo link.

Ara Vartanian - ara@blockchaingamealliance.org