Protocol Audit Report

Prepared by: Royal

Table of Contents

Protocol Summary

The DatingDapp protocol allows users to mint a soulbound NFT as their verified dating profile. Users can express interest in another user by paying 1 ETH to "like" their profile. If the like is mutual, the ETH paid (minus a 10% fee) is pooled into a shared multisig wallet, which both users can access for their first date. This system ensures genuine connections and meaningful, on-chain commitments.

Disclaimer

I made all effort to find as many vulnerabilities as possible within the given time frame. However, we hold no responsibility for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed, and the review of the code was solely focused on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low

		Impact		
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

The audit covered the following contracts:

- `LikeRegistry.sol`
- `MultiSig.sol`
- `SoulboundProfileNFT.sol`

Roles

- **Auditor**: Uche Royal
- **Client**: DatingDapp Team

Executive Summary

The audit identified several issues, including a high-severity issue related to the `userBalances` mapping not being updated, which renders the reward system ineffective. Additionally, a medium-severity reentrancy vulnerability was found in the `mintProfile` function of the `SoulboundProfileNFT` contract.

Issues Found

- **High**: 1 issue
- **Medium**: 1 issue
- **Low**: 0 issues
- **Informational**: 0 issues
- **Gas**: 0 issues

Findings

High

`userBalances` Is Never Updated – Matching Rewards Are Always Zero

Description: The `userBalances` mapping is never updated in the contract, meaning all balances remain at `0`. As a result, the `matchRewards` function will always compute a `totalRewards` of `0`.

Impact:

- Users never receive any rewards since `userBalances[from]` and `userBalances[to]` are always `0`.
- The `matchRewards` function becomes useless because it only distributes rewards that don’t exist.
- Total fees (`totalFees`) will always be `0`, making `withdrawFees` ineffective.

Proof of Concept: Nowhere in the contract is `userBalances[msg.sender]` updated. The `likeUser` function receives ETH but does not store it in `userBalances`.

```
function matchRewards(address from, address to) internal {
    uint256 matchUserOne = userBalances[from]; // Always 0
    uint256 matchUserTwo = userBalances[to];   // Always 0
    userBalances[from] = 0;
    userBalances[to] = 0;

    uint256 totalRewards = matchUserOne + matchUserTwo; // Always 0
    uint256 matchingFees = (totalRewards * FIXEDFEE) / 100; // Always 0
    uint256 rewards = totalRewards - matchingFees; // Always 0
    totalFees += matchingFees; // Always 0

    MultiSigWallet multiSigWallet = new MultiSigWallet(from, to);
    (bool success,) = payable(address(multiSigWallet)).call{value:
rewards}(""); // Sending 0 ETH
    require(success, "Transfer failed");
}
```

Recommended Mitigation: Update `userBalances` when a user likes another user:

```
function likeUser(address liked) external payable {
    require(msg.value >= 1 ether, "Must send at least 1 ETH");
    require(!likes[msg.sender][liked], "Already liked");
    require(msg.sender != liked, "Cannot like yourself");
    require(profileNFT.profileToToken(msg.sender) != 0, "Must have a
profile NFT");
    require(profileNFT.profileToToken(liked) != 0, "Liked user must have
a profile NFT");

    likes[msg.sender][liked] = true;
    emit Liked(msg.sender, liked);

    // ✅ Store ETH balance for future rewards
    userBalances[msg.sender] += msg.value;

    // ✅ Refund any excess ETH
    if (msg.value > 1 ether) {
        (bool success,) = payable(msg.sender).call{value: msg.value - 1 ether}("");
        require(success, "Refund failed");
    }

    // ✅ If mutual like, trigger rewards
    if (likes[liked][msg.sender]) {
        matches[msg.sender].push(liked);
        matches[liked].push(msg.sender);
        emit Matched(msg.sender, liked);
        matchRewards(liked, msg.sender);
    }
}
```

Medium

Reentrancy in `mintProfile` in SoulboundProfileNFT Contract

Description: The `mintProfile` function calls `_safeMint(msg.sender, tokenId)`, which triggers an external call to `onERC721Received`. If the recipient is a malicious smart contract, it can execute a reentrant call before `_profiles[tokenId]` and `profileToToken[msg.sender]` are updated.

Impact: An attacker can repeatedly reenter `mintProfile`, allowing them to mint multiple NFTs before the mapping `profileToToken[msg.sender]` is updated, violating the assumption that each address can have only one profile.

Proof of Concept: A malicious contract could look like this:

```
contract ReentrantAttacker {
    SoulboundProfileNFT target;

    constructor(address _target) {
        target = SoulboundProfileNFT(_target);
    }

    function attack(string memory name, uint8 age, string memory profileImage) external {
        target.mintProfile(name, age, profileImage);
    }

    function onERC721Received(
        address,
        address,
        uint256,
        bytes calldata
    ) external returns (bytes4) {
        if (address(target).balance > 0) {
            target.mintProfile("Reentered", 99, "ipfs://attack");
        }
        return this.onERC721Received.selector;
    }
}
```

Recommended Mitigation: Use the **Checks-Effects-Interactions** pattern by updating state variables before calling `_safeMint`:

```
function mintProfile(string memory name, uint8 age, string memory
profileImage) external {
    require(profileToToken[msg.sender] == 0, "Profile already exists");

    uint256 tokenId = ++_nextTokenId;

    // Update state before external call
    profileToToken[msg.sender] = tokenId;
    _profiles[tokenId] = Profile(name, age, profileImage);

    _safeMint(msg.sender, tokenId);

    emit ProfileMinted(msg.sender, tokenId, name, age, profileImage);
}
```

Low

No low-severity issues were identified.

Informational

No informational issues were identified.

Gas

No gas-related issues were identified.

Conclusion

The audit identified critical issues that need to be addressed to ensure the security and functionality of the DatingDapp protocol. The high-severity issue related to `userBalances` must be fixed to enable the reward system, and the medium-severity reentrancy vulnerability in `mintProfile` should be mitigated to prevent abuse. After these fixes, the protocol will be more secure and functional.