DatingDapp Audit Report

Prepared by Putfor

March 1, 2026

Table of Contents

Protocol Summary

DatingDapp is an on-chain social matching protocol composed of three contracts: `SoulboundProfileNFT`, `LikeRegistry`, and `MultiSigWallet`. Users create a non-transferable profile NFT, pay to like other profiles, and if likes are mutual, the protocol attempts to route pooled value (minus protocol fee) into a shared multisig wallet controlled by the matched pair.

The protocol's core trust assumptions are profile uniqueness, correct accounting of paid likes, and reliable movement of ETH between users, fee storage, and match wallets.

Disclaimer

Putfor 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 research by Putfor 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
	Low	M	M/L	L

I use the CodeHawks `https://docs.codehawks.com/hawks-auditors/how-to-evaluate-a-finding-severity` severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings in this document correspond to the following commit hash:

```
878bd34ef6607afe01f280cd5aedf3184fc4ca7b
```

Scope

```
src/
├── LikeRegistry.sol
├── MultiSig.sol
└── SoulboundProfileNFT.sol
```

Compatibilities

Blockchains:

- Ethereum / EVM-equivalent chains

Token standards and value flows:

- ERC721 (`SoulboundProfileNFT`)
- Native ETH flows in `LikeRegistry` and `MultiSigWallet`

Roles

Owner: Administrative role with privileged actions (for example, fee withdrawals and profile moderation paths).

User: Participant who mints a soulbound profile, likes other users, and participates in match-related fund flows.

Matched Pair: Two users that receive shared fund control through a dedicated multisig wallet.

Executive Summary

The review combined manual reasoning about protocol invariants with reproducible testing to validate exploitability. The methodology included line-by-line source analysis, adversarial scenario design, Foundry-based unit testing, and focused static review for dead code and unsafe state transitions.

The most significant findings affect core protocol guarantees: profile uniqueness can be bypassed through reentrancy in `mintProfile`, and the like-payment economic model is inconsistent with accounting and withdrawal logic in `LikeRegistry`. These high-severity issues are complemented by low-severity quality findings that reduce clarity and maintainability but are straightforward to remediate.

Issues found

Severity	Number of issues found
High	3
Low	3
Total	6

Findings

High

[H-1] Like Payments Are Never Accounted to User Balances, Breaking Core Match-Payout Logic

Description

`likeUser()` enforces a payment of at least `1 ETH` per like:

```
require(msg.value >= 1 ether, "Must send at least 1 ETH");
```

However, the paid value is never credited to either payer accounting or reward accounting.

The reward path (`matchRewards`) relies exclusively on `userBalances`:

```
uint256 matchUserOne = userBalances[from];
uint256 matchUserTwo = userBalances[to];
userBalances[from] = 0;
userBalances[to] = 0;

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

But `userBalances` is never increased anywhere in the contract. In practice, this means users still pay `>= 1 ETH` for each like, while mutual matches compute `totalRewards == 0` and send no meaningful value to the match multisig. The net effect is a direct contradiction of the documented economic model in `README.md`: paid likes do not translate into match rewards.

Risk

Likelihood: High

The issue is deterministic and always present in current logic: every like payment follows this path, and no alternative code path credits `userBalances`.

Impact: High

The protocol's primary economic mechanism is broken. Users pay substantial value (`>=1 ETH`) but mutual matches do not receive pooled rewards as described. This can lead to severe user loss of confidence and economic failure of the product.

Proof of Concept

Static flow proof in `src/LikeRegistry.sol`:

1. `likeUser()` requires payment (`msg.value >= 1 ether`).
2. `likeUser()` does not write to `userBalances`.
3. `matchRewards()` computes payout only from `userBalances[from] + userBalances[to]`.
4. Since both are never funded by protocol logic, `totalRewards` remains zero.

Consequence: users pay ETH for likes, but match payout logic has no funded inputs.

Recommended Mitigation

Track each payer's like deposits and consume those tracked balances when a match is formed.

Illustrative patch:

```
 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");

+   userBalances[msg.sender] += msg.value;

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

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

Also consider documenting whether users can accumulate multiple deposits before matching and how partial refunds/withdrawals should work.

[H-2] Untracked ETH (Like Payments and Direct Transfers) Can Become Permanently Stuck

Description

The contract accepts ETH through paid likes (`likeUser()`) and through direct transfers (`receive()`).

```
function likeUser(address liked) external payable { ... }
receive() external payable {}
```

However, the only withdrawal path is `withdrawFees()`, and that function transfers only `totalFees`:

```
uint256 totalFeesToWithdraw = totalFees;
totalFees = 0;
(bool success,) = payable(owner()).call{value: totalFeesToWithdraw}("");
```

There is no function that withdraws `address(this).balance`, and there is no complete accounting path that maps all incoming ETH to `totalFees`. In the current logic, ETH paid in `likeUser()` is not added to `totalFees`, and ETH received through `receive()` is not added either. As a consequence, contract ETH can accumulate outside the withdrawable fee bucket and become permanently unreachable.

Risk

Likelihood: High

The issue follows normal protocol behavior (users paying for likes) and does not require edge conditions or privileged access.

Impact: High

User ETH and externally sent ETH can become permanently locked, causing irreversible value loss and significant operational and reputational damage.

Proof of Concept

Deterministic scenario:

1. User calls `likeUser()` and pays `1 ETH`.
2. Contract balance increases by `1 ETH`.
3. `totalFees` does not increase from that payment path.
4. Owner calls `withdrawFees()` and can withdraw only the `totalFees` amount.

5. Remaining ETH (from likes/direct sends) has no dedicated withdrawal path and can remain stuck indefinitely.

Recommended Mitigation

Define one coherent accounting model and align all inflows and outflows to that model.

Minimum safe approach:

- Account like payments either to user escrow (`userBalances`) or to fee pool (`totalFees`) according to intended economics.
- Restrict or remove raw `receive()` if unsolicited ETH is not intended.
- Add an explicit recovery path for unexpected ETH with strict access control and event logging.

Illustrative addition for accidental ETH recovery:

```diff
+event ExcessEthRecovered(address indexed to, uint256 amount);
+
+function recoverExcessEth(uint256 amount) external onlyOwner {
+    require(amount <= address(this).balance, "Insufficient balance");
+    (bool success,) = payable(owner()).call{value: amount}("");
+    require(success, "Transfer failed");
+    emit ExcessEthRecovered(owner(), amount);
+}
```

If user funds are intended to be non-custodial and match-bound, replace owner recovery with explicit user-facing refund/claim mechanics.

[H-3] Reentrancy in `mintProfile()` Allows Multiple Profile NFTs for the Same Address

Description

`mintProfile()` checks `profileToToken[msg.sender] == 0` and then calls `_safeMint(msg.sender, tokenId)` before updating `profileToToken[msg.sender]`.

```
function mintProfile(string memory name, uint8 age, string memory profileImage)
external {
    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;
}
```

Because `_safeMint` performs an external callback to `onERC721Received` when `msg.sender` is a contract, an attacker contract can reenter `mintProfile()` before `profileToToken[msg.sender]` is set. During reentry, the same precondition still passes, allowing repeated mints for one address. This breaks the one-profile-per-user invariant that the contract enforces for EOAs and intended usage.

Risk

Likelihood: High

The attack path is straightforward and deterministic for contract callers implementing `onERC721Received`. It does not require privileged access or unusual chain conditions.

Impact: High

The core identity constraint of the protocol is violated: a single participant can mint multiple soulbound profiles. This undermines profile uniqueness, match integrity, and trust assumptions in downstream logic that treats one address as one profile identity.

Proof of Concept

The PoC is implemented in `test/testSoulboundProfileNFT.t.sol` using a helper attacker contract `ReentrantMintReceiver` and the test `test_ReentrancyAllowsMultipleProfilesForSameAddress()`.

```
contract ReentrantMintReceiver {
    SoulboundProfileNFT private immutable target;
    uint256 private immutable maxMints;
    uint256 private mintCount;

    constructor(SoulboundProfileNFT _target, uint256 _maxMints) {
        target = _target;
        maxMints = _maxMints;
    }

    function executeMintAttack() external {
        target.mintProfile("Mallory", 42, "ipfs://attacker");
    }

    function onERC721Received(address, address, uint256, bytes calldata)
external returns (bytes4) {
        mintCount++;
        if (mintCount < maxMints) {
            target.mintProfile("Mallory", 42, "ipfs://attacker");
        }
        return this.onERC721Received.selector;
    }
}

function test_ReentrancyAllowsMultipleProfilesForSameAddress() public {
    ReentrantMintReceiver attacker = new ReentrantMintReceiver(soulboundNFT, 5);

    attacker.executeMintAttack();
    assertEq(soulboundNFT.balanceOf(address(attacker)), 5, "Attacker should own
5 NFTs");
    assertEq(soulboundNFT.ownerOf(1), address(attacker), "Token 1 owner
mismatch");
    assertEq(soulboundNFT.ownerOf(2), address(attacker), "Token 2 owner
mismatch");
    assertEq(soulboundNFT.ownerOf(3), address(attacker), "Token 3 owner
mismatch");
    assertEq(soulboundNFT.ownerOf(4), address(attacker), "Token 4 owner
mismatch");
    assertEq(soulboundNFT.ownerOf(5), address(attacker), "Token 5 owner
mismatch");
}
```

Run the test with:

```
forge test --match-test test_ReentrancyAllowsMultipleProfilesForSameAddress -vv
```

Output:

```
Ran 1 test for test/testSoulboundProfileNFT.t.sol:SoulboundProfileNFTTest
[PASS] test_ReentrancyAllowsMultipleProfilesForSameAddress() (gas: 941176)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.55ms (532.54µs
CPU time)

Ran 1 test suite in 84.06ms (5.55ms CPU time): 1 tests passed, 0 failed, 0
skipped (1 total tests)
```

Thus, the PoC demonstrates that a single address can mint multiple profile NFTs via reentrancy.

Recommended Mitigation

Apply checks-effects-interactions ordering so the profile ownership state is committed before external interaction. In practice, write `profileToToken[msg.sender]` before `_safeMint`, then mint and store metadata. This causes any reentrant `mintProfile()` call to fail immediately at the `Profile already exists` check.

Illustrative patch:

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

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

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

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

For defense in depth, `nonReentrant` can be added as an additional guard, but the state-order fix above is the critical correction.

Low

[L-1] Unused Custom Error `NotEnoughApprovals` in `MultiSigWallet`

Description

`MultiSigWallet` declares a custom error:

```
error NotEnoughApprovals();
```

but the approval check in `executeTransaction()` uses a string-based `require` instead:

```
require(txn.approvedByOwner1 && txn.approvedByOwner2, "Not enough approvals");
```

As a result, `NotEnoughApprovals` is dead code. This creates inconsistency in error handling style and slightly increases maintenance overhead.

Risk

Likelihood: High

The issue is present unconditionally in the current source code.

Impact: Low

No direct security compromise is introduced. The impact is limited to code quality, consistency, and minor gas/readability inefficiency.

Proof of Concept

Static check shows declaration without usage:

```
rg -n "NotEnoughApprovals|Not enough approvals" src/MultiSig.sol
```

Expected output pattern:

```
src/MultiSig.sol:7:error NotEnoughApprovals();
src/MultiSig.sol:72:require(txn.approvedByOwner1 && txn.approvedByOwner2, "Not
enough approvals");
```

This confirms the custom error exists but is never used.

Recommended Mitigation

Use the custom error in the approval check for consistency and lower revert-cost footprint:

```diff
- require(txn.approvedByOwner1 && txn.approvedByOwner2, "Not enough approvals");
+ if (!(txn.approvedByOwner1 && txn.approvedByOwner2)) revert
NotEnoughApprovals();
```

If string-based errors are preferred project-wide, remove the unused custom error declaration instead.

[L-2] Unused `Like` Struct in `LikeRegistry`

Description

`LikeRegistry` defines a `Like` struct:

```
struct Like {
    address liker;
    address liked;
    uint256 timestamp;
}
```

However, the contract never instantiates, stores, or returns this struct. The active like/match flow relies on `likes` mapping and `matches` arrays instead.

This leaves `Like` as unused code, which can mislead reviewers about expected storage design and increases code noise.

Risk

Likelihood: High

The unused declaration is always present in the current implementation.

Impact: Low

This is a maintainability and clarity issue, not a direct exploitable vulnerability.

Proof of Concept

Static usage check:

```
rg -n "struct Like|Like\b" src/LikeRegistry.sol
```

Expected output pattern shows only the declaration location and no operational usage.

```
10:    struct Like {
```

Recommended Mitigation

Remove the unused struct if it is not part of near-term functionality:

```
- struct Like {
-     address liker;
-     address liked;
-     uint256 timestamp;
- }
```

If historical like records are planned, implement full storage and retrieval paths that actually use this structure.

[L-3] `FIXEDFEE` Should Be `constant` Instead of `immutable`

Description

`LikeRegistry` declares:

```
uint256 immutable FIXEDFEE = 10;
```

`FIXEDFEE` is assigned at declaration time and never receives constructor input. In this pattern, `constant` is a more appropriate qualifier because the value is compile-time fixed.

Using `constant` makes intent clearer and can improve bytecode/runtime efficiency compared with a non-constructor `immutable` literal.

Risk

Likelihood: High

The declaration pattern is present in the current implementation.

Impact: Low

No direct security issue is introduced; this is a code-quality and gas-optimization finding.

Proof of Concept

Relevant lines:

- declaration: `uint256 immutable FIXEDFEE = 10;`
- usage: `uint256 matchingFees = (totalRewards * FIXEDFEE) / 100;`

There is no constructor parameter or runtime update path for `FIXEDFEE`.

Recommended Mitigation

Change `FIXEDFEE` to `constant`:

```diff
- uint256 immutable FIXEDFEE = 10;
+ uint256 constant FIXEDFEE = 10;
```

Optionally expose it as `public constant` if external visibility is needed for integrations or frontends.