Protocol DatingApp Audit Report

Version 1.0

AGAMSO

February 12, 2025

DatingApp

AGMASO

February 12, 2025

Prepared by: [AGMASO] Lead Auditors: - Alejandro G.

Table of Contents

- Table of Contents
- Disclaimer
- Risk Classification
- Audit Details
  - Scope
- Protocol Summary
  - Roles
- Executive Summary
  - Issues Found
  - High
    * [H-1] Lack of UniswapV2 slippage protection in `UniswapAdapter::_uniswapInvest` enables frontrunners to steal profits
    * [H-2] `ERC4626::totalAssets` checks the balance of vault's underlying asset even when the asset is invested, resulting in incorrect values being returned
    * [H-3] Guardians can infinitely mint `VaultGuardianToken`s and take over DAO, stealing DAO fees and maliciously setting parameters
  - Low
    * [L-1] Incorrect vault name and symbol
    * [L-2] Unassigned return value when divesting AAVE funds

Disclaimer

The AGMASO 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
	Low	M	M/L	L

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

Audit Details

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

Scope

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

Protocol Summary

Roses are red, violets are blue, use this DatingDapp and love will find you.! Dating Dapp lets users mint a soulbound NFT as their verified dating profile. To express interest in someone, they pay 1 ETH to "like" their profile. If the like is mutual, all their previous like payments (minus a 10% fee) are pooled into a shared multisig wallet, which both users can access for their first date. This system ensures genuine connections, and turns every match into a meaningful, on-chain commitment.

Roles

Executive Summary

Issues Found

Severity	Number of issues found
High	1
Medium	2
Low	11
Info	1
Gas	1
Total	16

High

[H-1] Lack of Tracking for the `userBalances` Mapping in `LikeRegistry.sol` Disrupts the Protocol's Functionality

Description:

In `LikeRegistry.sol`, we find the core logic of the protocol, where users can like each other and get matched. To perform a like, a user must pay at least **1 ETH** by calling the function `likeUser(address liked)`.

A crucial part of the protocol is the **tracking of user payments** using the `userBalances()` mapping. This mapping records the amount of ETH each user has paid, which is later used to create an

instance of `MultiSigWallet.sol` and transfer the correct funds from both matched users to the MultisigWallet.

The issue arises because **`LikeRegistry::likeUser()` does not update the `userBalances` mapping**. As a result, the stored balance remains **zero**, even if users pay ETH to the protocol. This flaw has **two major consequences**:

1. **Locked Funds** – The ETH sent by users **cannot be accessed**, as the protocol relies on `userBalances` to track amounts for `MultiSigWallet` transfers.
2. **Fee Withdrawal Failure** – The protocol calculates fees based on `userBalances`, meaning that if the balance remains **zero**, the fees will also be **zero**, preventing the contract owner from withdrawing them.

Ultimately, this **breaks the entire financial flow of the protocol**, leading to **permanently locked funds** and an **inoperable fee system**.

```
1
2  function likeUser(address liked) external payable {
3
4          require(msg.value >= 1 ether, "Must send at least 1 ETH");
5
6          require(!likes[msg.sender][liked], "Already liked");
7
8          require(msg.sender != liked, "Cannot like yourself");
9
10         require(
11             profileNFT.profileToToken(msg.sender) != 0,
12             "Must have a profile NFT"
13         );
14         require(
15             profileNFT.profileToToken(liked) != 0,
16             "Liked user must have a profile NFT"
17         );
18
19         likes[msg.sender][liked] = true;
20 @>      //Missing update of mapping userBalances();
21         emit Liked(msg.sender, liked);
22
23         if (likes[liked][msg.sender]) {
24             matches[msg.sender].push(liked);
25             matches[liked].push(msg.sender);
26             emit Matched(msg.sender, liked);
27             matchRewards(liked, msg.sender);
28         }
29     }
```

Impact:

The impact of this issue is **critical**, as the funds paid by users will be **permanently locked**, and the

core functionalities of the protocol will be disrupted.

The **likelihood** of this issue occurring is also **high**, as it will consistently affect every transaction.

Proof of Concept:

PoC

```

function test__audit__noTrackingOfUserBalancesCriticalError() public {
        //Minting nfts for each user
        vm.prank(user); // Simulates user calling the function
        soulboundNFT.mintProfile("Alice", 25, "ipfs://profileImage");
        vm.prank(user2); // Simulates user calling the function
        soulboundNFT.mintProfile("David", 26, "ipfs://profileImage");

        //user will like user2
        vm.prank(user);
        likeRegistry.likeUser{value: 1 ether}(address(user2));
        uint256 balancesOfUserPayed = likeRegistry.userBalances(address
            (user));
        console.log(
            "Balance of the userBalance(user) after first payment: ",
            balancesOfUserPayed
        );
        uint256 balanceOfCOntract = (address(likeRegistry)).balance;
        console.log(
            "LikeRegistry balance after first payment: ",
            balanceOfCOntract
        );

        assert(likeRegistry.likes(address(user), address(user2)) ==
            true);

        //user2 will like user
        vm.prank(user2);
        likeRegistry.likeUser{value: 1 ether}(address(user));
        uint256 balanceOfCOntractAfterSendingToMultisig = address(
            likeRegistry)
            .balance;

        // Balance of LikeRegistry SC is still 2 ether. Meaning the
            lack of tracking userBlances mapping is
        // breaking the protocol.
        console.log(
            "LikeRegistry balance after creatingand sending to multisig
                : ",
            balanceOfCOntractAfterSendingToMultisig
        );

        // Not passing this test, because the lack of tracking
            userBlance breaks the Protocol
```

```
39          assert(balanceOfCOntractAfterSendingToMultisig <
               balanceOfCOntract);
40      }
```

Recommended Mitigation:

To mitigate this issue we encourage you to add an update of the `userBalances()` mapping inside the `LikeRegistry::likeUser()` function.

```
1
2   function likeUser(address liked) external payable {
3           require(msg.value >= 1 ether, "Must send at least 1 ETH");
4
5           require(!likes[msg.sender][liked], "Already liked");
6
7           require(msg.sender != liked, "Cannot like yourself");
8
9           require(
10              profileNFT.profileToToken(msg.sender) != 0,
11              "Must have a profile NFT"
12          );
13          require(
14              profileNFT.profileToToken(liked) != 0,
15              "Liked user must have a profile NFT"
16          );
17
18          likes[msg.sender][liked] = true;
19  +       userBalances[msg.sender] += msg.value;
20          emit Liked(msg.sender, liked);
21
22
23          if (likes[liked][msg.sender]) {
24              // e pushea tanto en uno como en otro array en mathces
                   mapping
25              matches[msg.sender].push(liked);
26              matches[liked].push(msg.sender);
27
28              emit Matched(msg.sender, liked);
29              matchRewards(liked, msg.sender);
30          }
31      }
```

Medium

[M-1] Lack of Deployment Script leads to a possible Unauthorized Contract Deployment of `LikeRegistry.sol` from a malicious user getting access to ownable functions such as `LikeRegistry.sol::withdrawFees`

Description:
The project scope does not provide a deployment script, leaving the responsibility of deploying critical contracts to the protocol deployer. If the deployer forgets to deploy the `LikeRegistry.sol` contract, a malicious user could deploy it first, making themselves the contract owner and gaining control over critical functions. This exposes the protocol to unauthorized fee withdrawals and potential fund theft.

Impact:
The impact is high, as an attacker could deploy the `LikeRegistry` contract before the legitimate deployer and become the owner. This would allow the attacker to withdraw all accumulated fees using the `withdrawFees()` function. The application's revenue model would be compromised, leading to financial losses for the protocol.

The likelihood is medium. We are assuming that the script might fail to deploy the `LikeRegistry.sol` contract. This assumption is based on the fact that in the test file, the protocol developer has not initialized this contract, making us think that they might also forget to include it in the deployment script. This scenario would only occur if the deployer forgets to include it.

Proof of Concept:
1. The `LikeRegistry` contract does not enforce an ownership restriction on deployment. 2. Any user can deploy the contract using: `solidity LikeRegistry maliciousRegistry = new LikeRegistry(address(_profileNFT));`

PoC

```
```javascript

contract SoulboundProfileNFTTest is Test {
SoulboundProfileNFT soulboundNFT;
LikeRegistry likeRegistry;
address user = address(0x123);
address user2 = address(0x456);
address owner = address(this); // Test contract acts as the owner

function setUp() public {
    soulboundNFT = new SoulboundProfileNFT();

    vm.deal(user, 100 ether);
    vm.deal(user2, 100 ether);
```
```

```
}
function test__audit__userGainsOwnershipOfLikeRegistry() public {
    vm.prank(user); // Simulates user calling the function
    soulboundNFT.mintProfile("Alice", 25, "ipfs://profileImage");
    vm.prank(user2); // Simulates user calling the function
    soulboundNFT.mintProfile("David", 26, "ipfs://profileImage");

    vm.prank(user);
    LikeRegistry likeRegistryOwnedByUser = new LikeRegistry(
        address(soulboundNFT)
    );

    vm.expectRevert();
    likeRegistryOwnedByUser.withdrawFees();

    vm.prank(user);
    likeRegistryOwnedByUser.withdrawFees();
}
}

```
```

Recommended Mitigation: Please don't forget to deploy `LikeRegistry.sol` immediately after deploying `SoulboundProfileNFT.sol`.

[M-2] Missing Require Statement to Limit Submission of Transactions with a Value Greater Than Contract Funds in `MultiSigWallet::submitTransaction()`, Leading to Future Failed Transaction Execution.

Description:

The function `MultiSigWallet::executeTransaction(uint256 _txId)` allows the execution of approved transactions from a multisig wallet. However, there is no validation at the time of transaction creation to ensure that the requested `value` does not exceed the available funds in the multisig wallet. This can result in transactions being approved but failing at the execution stage due to insufficient funds.

```

function submitTransaction(
        address _to,
        uint256 _value
    ) external onlyOwners {
        if (_to == address(0)) revert InvalidRecipient();
        if (_value == 0) revert InvalidAmount();
@>      // Missing check
        transactions.push(Transaction(_to, _value, false, false, false)
            );
```
```

```
10
11          uint256 txId = transactions.length - 1;
12          emit TransactionCreated(txId, _to, _value);
13      }
```

Impact:

- Transactions that exceed the wallet's balance will fail upon execution, leading to unnecessary transaction approvals and wasted gas costs.
- Users may approve transactions assuming they will be executed successfully, only to encounter execution failures.
- This could result in an inefficient user experience and additional operational overhead in reat-tempting transactions with adjusted values.

Proof of Concept:

1. Assume the multisig wallet has **3 ETH** in balance.
2. A transaction is created with `txn.value = 5 ETH`.
3. The transaction gets approved by both owners.
4. When calling `executeTransaction(_txId)`, it reaches the following line:

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

5. It will revert, as the value is greater than the multisig's funds.

Recommended Mitigation:

Add a validation check at the moment of transaction creation to prevent transactions with a value greater than the available balance in the multisig wallet.

```
+       error AmountToBig(uint256 value);

function submitTransaction(
        address _to,
        uint256 _value
    ) external onlyOwners {
        if (_to == address(0)) revert InvalidRecipient();
        if (_value == 0) revert InvalidAmount();
```

```
+       if (_value > address(this).balance) revert AmountToBig(_value);
        transactions.push(Transaction(_to, _value, false, false, false)
            );
        // e el txId sera igual al puesto en el array. por esto el
            lenght - 1
        uint256 txId = transactions.length - 1;
        emit TransactionCreated(txId, _to, _value);
    }
```

Low

[L-1]: Centralization Risk for trusted owners

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds.

4 Found Instances

- Found in src/LikeRegistry.sol Line: 11

```
contract LikeRegistry is Ownable {
```

- Found in src/LikeRegistry.sol Line: 122

```
    function withdrawFees() external onlyOwner {
```

- Found in src/SoulboundProfileNFT.sol Line: 10

```
contract SoulboundProfileNFT is ERC721, Ownable {
```

- Found in src/SoulboundProfileNFT.sol Line: 77

```
    function blockProfile(address blockAddress) external onlyOwner
        {
```

[L-2]: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

3 Found Instances

- Found in src/LikeRegistry.sol Line: 4

```
pragma solidity ^0.8.19;
```

- Found in src/MultiSig.sol Line: 2

```
1 pragma solidity ^0.8.19;
```

- Found in src/SoulboundProfileNFT.sol Line: 2

```
1 pragma solidity ^0.8.19;
```

[L-3]: Missing checks for `address(0)` when assigning values to address state variables

Check for `address(0)` when assigning values to address state variables.

1 Found Instances

- Found in src/LikeRegistry.sol Line: 49

```
1         profileNFT = SoulboundProfileNFT(_profileNFT);
```

[L-4]: `public` functions not used internally could be marked `external`

Instead of marking a function as **`public`**, consider marking it as `external` if it is not used internally.

3 Found Instances

- Found in src/SoulboundProfileNFT.sol Line: 93

```
1     function transferFrom(address, address, uint256) public pure
         override {
```

- Found in src/SoulboundProfileNFT.sol Line: 98

```
1     function safeTransferFrom(
```

- Found in src/SoulboundProfileNFT.sol Line: 112

```
1     function tokenURI(
```

[L-5]: Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three

or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

4 Found Instances

- Found in src/MultiSig.sol Line: 27

```
1      event TransactionCreated(
```

- Found in src/MultiSig.sol Line: 33

```
1      event TransactionExecuted(
```

- Found in src/SoulboundProfileNFT.sol Line: 27

```
1      event ProfileMinted(
```

- Found in src/SoulboundProfileNFT.sol Line: 34

```
1      event ProfileBurned(address indexed user, uint256 tokenId);
```

[L-6]: PUSH0 is not supported by all chains

Solc compiler version 0.8.20 switches the default target EVM version to Shanghai, which means that the generated bytecode will include PUSH0 opcodes. Be sure to select the appropriate EVM version in case you intend to deploy on a chain other than mainnet like L2 chains that may not support PUSH0, otherwise deployment of your contracts will fail.

3 Found Instances

- Found in src/LikeRegistry.sol Line: 4

```
1  pragma solidity ^0.8.19;
```

- Found in src/MultiSig.sol Line: 2

```
1  pragma solidity ^0.8.19;
```

- Found in src/SoulboundProfileNFT.sol Line: 2

```
1  pragma solidity ^0.8.19;
```

[L-7]: Internal functions called only once can be inlined

Instead of separating the logic into a separate function, consider inlining the logic into the calling function. This can reduce the number of function calls and improve readability.

1 Found Instances

- Found in src/LikeRegistry.sol Line: 89

```
1       function matchRewards(address from, address to) internal {
```

[L-8]: Unused Custom Error

it is recommended that the definition be removed when custom error is unused

1 Found Instances

- Found in src/MultiSig.sol Line: 8

```
1       error NotEnoughApprovals();
```

[L-9]: State variable changes but no event is emitted.

State variable changes in this function but no event is emitted.

1 Found Instances

- Found in src/LikeRegistry.sol Line: 122

```
1       function withdrawFees() external onlyOwner {
```

[L-10]: State variable could be declared immutable

State variables that are should be declared immutable to save gas. Add the `immutable` attribute to state variables that are only changed in the constructor

3 Found Instances

- Found in src/LikeRegistry.sol Line: 22

```
1       SoulboundProfileNFT public profileNFT;
```

- Found in src/MultiSig.sol Line: 13

```
1       address public owner1;
```

- Found in src/MultiSig.sol Line: 14

```
1       address public owner2;
```

[L-11] Dead Code in `LikeRegistry.sol` – Unused `Like` Struct

Description:

The `Like` struct is declared in `LikeRegistry.sol`, but it is never used within the contract. This results in unnecessary dead code, which increases contract size and reduces code clarity without providing any functional value. Unused structures can also lead to confusion for future maintainers and auditors, as they may assume the struct is intended for use in some part of the logic.

```
1
2  contract LikeRegistry is Ownable {
3      // Struct to store information about a "Like" action
4      struct Like {
5          address liker;
6          address liked;
7          uint256 timestamp;
8      }
9      ...
10 }
```

Recommended Mitigation:

Remove the unused struct if it is not needed to reduce contract size and improve maintainability.

Refactor the contract to utilize the struct if it was initially intended to store like-related data. This could improve readability and reduce redundant mappings or variables if applicable.

[I-1] Internal Functions Should Follow Naming Conventions (Use Leading Underscore)

Description:

In Solidity, internal functions should follow the convention of using a leading underscore (_) in their names. This helps distinguish them from external or public functions, improving readability and maintainability. In `LikeRegistry.sol`, the function `matchRewards` is declared as `internal`, but it does not follow this naming convention.

```
1  function matchRewards(address from, address to) internal {}
```

Recommended Mitigation:

To align with best practices and improve code clarity, add an underscore (_) at the beginning of the function name.

```
1
2  - function matchRewards(address from, address to) internal {}
3  + function _matchRewards(address from, address to) internal {}
```

[G-1] Inefficient Storage Usage in `MultiSigWallet::approveTransaction()` (Direct Storage Modification -> Unnecessary Gas Costs)

Description:

In the `approveTransaction(uint256 _txId)` function, a `Transaction` struct is directly referenced in storage using:

```
1
2  function approveTransaction(uint256 _txId) external onlyOwners {
3
4          require(_txId < transactions.length, "Invalid transaction ID");
5  @>      Transaction storage txn = transactions[_txId];
6          require(!txn.executed, "Transaction already executed");
7
8          if (msg.sender == owner1) {
9              if (txn.approvedByOwner1) revert AlreadyApproved();
10             txn.approvedByOwner1 = true;
11         } else {
12             if (txn.approvedByOwner2) revert AlreadyApproved();
13             txn.approvedByOwner2 = true;
14         }
15
16         emit TransactionApproved(_txId, msg.sender);
17
18     }
```

Since all modifications to txn are reflected directly in storage, every update incurs high gas costs. Instead, copying the struct to memory, making the necessary modifications, and then writing it back to storage at the end can significantly reduce gas consumption.

Impact

- Increased gas costs due to multiple writes to storage within the function.
- Inefficient state modification leads to unnecessary on-chain storage operations.
- Can be optimized to reduce execution costs, making transactions more affordable for users.

Recommended Mitigation

```
1
2  function approveTransaction(uint256 _txId) external onlyOwners {
3      require(_txId < transactions.length, "Invalid transaction ID");
4      require(!transactions[_txId].executed, "Transaction already
           executed");
5  -   Transaction storage txn = transactions[_txId];
6  +   Transaction memory txn = transactions[_txId];
7
8      if (msg.sender == owner1) {
9          if (txn.approvedByOwner1) revert AlreadyApproved();
```

```
10          txn.approvedByOwner1 = true;
11      } else {
12          if (txn.approvedByOwner2) revert AlreadyApproved();
13          txn.approvedByOwner2 = true;
14      }
15
16  +   transactions[_txId] = txn; // Single storage write to save gas
17
18      emit TransactionApproved(_txId, msg.sender);
19  }
```