Palm Network

Date	May 2023

1 Executive Summary

This report presents the results of our engagement with **Palm Network** to review **FactoryDAO Protocol**.

The review was conducted over a single week by a single security engineer, from **May 15th, 2023** to **May 19th, 2023** A total of 5 person-days were spent.

The audit was made on a best-effort basis, since the original scope for the audit was too large to be covered in a single week, we decided together with the client to focus the security assessment on the **Bank** repository, while as for the rest of the scope to focus more on the quality of the code.

Overall, our assessment of the code quality left us with a positive impression. It is evident that the team has effectively implemented robust security practices, particularly when it comes to securing smart contracts. No major security issues were found during the engagement, however, it is important to mention again that this report should not be counted upon as the only security assessment before deployment mainly due to the engagement time constraints. It is highly recommended to have an additional security review for all smart contracts before deployment.

2 Scope

Our review focused on the commit hash for the **Bank** repository `ead71e407526bd7918354eb38435cebc7b6416cb` and on `2c5b217f0a5d91bda679dd988b1a2f647935bbd7` for the **Yield** repository. The list of files in scope can be found in the Appendix.

2.1 Objectives

Together with the **Palm Network** team, we identified the following priorities for our review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 Recommendations

3.1 Implementing "in-house" libraries instead of importing well-vetted open source libraries is not recommended

Description

1. `MerkleLib` is an "in-house" library that is used to verify that a leaf is part of a merkle tree. Although we were not able to find any concrete issues in the current implementation, it is recommended to use a well-vetted open source library like MerkleProof.sol instead.
2. `SafeMathLib` is an "in-house" library that is used to ensure safe arithmetic operations. Although we were not able to find any concrete issues in the current implementation, it is recommended to use a well-vetted open source library like SafeMath.sol instead, or alternatively consider using compiler version 0.8.x instead.
3. `Token` is an "in-house" ERC20 token contract. Although we were not able to find any concrete issues in the current implementation, it is recommended to use a well-vetted open source library like ERC20.sol

Examples

code/contracts/MerkleLib.sol:L5-L23

```
library MerkleLib {

    function verifyProof(bytes32 root, bytes32 leaf, bytes32[] calldata proof) public pure returns (bool) {
        bytes32 currentHash = leaf;

        uint proofLength = proof.length;
        for (uint i; i < proofLength;) {
            currentHash = parentHash(currentHash, proof[i]);
            unchecked { ++i; }
        }

        return currentHash == root;
    }

    function parentHash(bytes32 a, bytes32 b) private pure returns (bytes32) {
        return keccak256(a < b ? abi.encode(a, b) : abi.encode(b, a));
    }

}
```

3.2 `MultiSender.multisend` - Consider using `.call` instead of `transfer` and log the failed recipient address in case of failure

Description

The `Multisend.multisend` function serves the purpose of distributing ether to multiple addresses in a single transaction. The function uses the native function of `.transfer` to transfer ether to the recipients. In case one of the recipients is a contract that denies the recieving of ether, the entire transaction will fail, which in itself is not an issue since the transaction sender can specify a different array of recipients instead, but in the current version of the code, it will require a debugging of the transaction which can be easier if `.call` will be used instead, and in case of failure, the failed recipient address will be included in the error message.

The `Multisend.multisend` function plays a crucial role in facilitating the distribution of ether to multiple addresses within a single transaction. This is achieved through the utilization of the `.transfer` native function for transferring ether to the intended recipients. However, it's worth noting that if one of the recipients happens to be a contract that rejects incoming ether, the entire transaction will fail. Although this is not inherently problematic, as the transaction sender can specify an alternative array of recipients, the current version of the code requires manual debugging of the transaction in such cases.

To enhance the debugging process and provide more helpful error messages, it would be beneficial to replace `.transfer` with `.call`. By doing so, in the event of a failure, the error message can include the address of the recipient that caused the issue.

Examples

code/contracts/MultiSend.sol:L9-L21

```
    function multisend(address payable[] memory addresses, uint256 amount) payable public {
        uint256 total = msg.value;

        for (uint i=0; i < addresses.length; i++) {
            // the total should be greater than the sum of the amounts
            require(total >= amount, "The value is not sufficient");
            total -= amount;

            // send the specified amount to the recipient
            addresses[i].transfer(amount);
        }
    }
}
```

3.3 `Token.freeze` - `FrozenTokens.id` is never used

Description

The `Token.freeze` function stores the `numFrozenStructs` inside the `id` field of `FrozenTokens`, but this value is never used throughout the rest of the contract, and thus might be removed to ensure a more gas-efficient implementation.

Examples

code/contracts/Token.sol:L68

```
frozenTokensMap[numFrozenStructs] = FrozenTokens(numFrozenStructs, block.timestamp, freezeDays, amount, true, msg.sender);
```

3.4 `MerkleResistor.verifyVestingSchedule` - Inconsistent inline comments and implementaion of `MerkleTree storage tree`

Description

The `verifyVestingSchedule` function refers to `merkleTrees[treeIndex]` and holds a `storage` pointer named `tree` while the inline comment suggests that it should be a `memory` pointer instead.

Examples

code/contracts/MerkleResistor.sol:L277-L278

```
// memory not storage, since we do not edit the tree, and it's a view function anyways
MerkleTree storage tree = merkleTrees[treeIndex];
```

Recommendation

Consider resolving the described inconsistency.

3.5 `BasicPoolFactory,PermissionlessBasicPoolFactory` Inconsistent usage of custom errors and error strings

Description

`BasicPoolFactory` uses error strings while `PermissionlessBasicPoolFactory` uses custom errors. It is highly recommended to be consistent with the error handling policy in the repository, both for maintenance purposes and for gas efficiency.

Examples

code/contracts/BasicPoolFactory.sol:L137

```
require(success, 'Token transfer failed');
```

code/contracts/PermissionlessBasicPoolFactory.sol:L239

```
revert UninitializedPool(poolId);
```

3.6 Review the Code Quality recommendations in Appendix 1

Other comments related to readability and best practices are listed in Appendix 1

4 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 `BasicPoolFactory/PermissionlessBasicPoolFactory.withdrawExcessRewards` - depositors may leave tiny amounts of tokens intentionally in order to lock excess rewards Medium

Description

`withdrawExcessRewards` is used to withdraw any excess reward tokens left in the contract after the pool reached maturity (`blocktimestamp > pool.endTime`). Calls to `withdrawExcessRewards` will revert in case there are still deposits left in the pool. malicious depositors can use it in certain cases to cause an infinite lock of rewards by leaving tiny amount of tokens in the contract intentionally, in case the value of the rewards is higher than the tiny amount of tokens, this attack will be effective.

Examples

code/contracts/BasicPoolFactory.sol:L170

```
require(pool.totalDepositsWei == 0, 'Cannot withdraw until all deposits are withdrawn');
```

Recommendation

Consider changing the `withdrawExcessRewards` function so that even if there are some deposits left, in case an additional period has passed, it will be possible for anyone to withdraw the excess rewards only.

4.2 `BasicPoolFactory.withdraw` - Users will not be able to withdraw their deposited tokens in case there are not enough rewards Medium

Description

`BasicPoolFactory.withdraw` facilitates the withdraw process for depositors. rewards are being withdrawn first, and then the deposit itself. All `.transfer` operations must succeed in order for the entire transaction to succeed. However, in the current version of the code, it is hard to tell whether the required reward amounts that are computed in `getRewards` will actually be in the contract, and therefore the rewards transfers might fail, which will cause the entire transaction to fail, which means users deposits will be effectively locked.

Examples

code/contracts/BasicPoolFactory.sol:L142-L165

```
function withdraw(uint poolId, uint receiptId) external {
    Pool storage pool = pools[poolId];
    require(pool.id == poolId, 'Uninitialized pool');
    Receipt storage receipt = pool.receipts[receiptId];
    require(receipt.id == receiptId, 'Can only withdraw real receipts');
    require(receipt.owner == msg.sender || block.timestamp > pool.endTime, 'Can only withdraw your own deposit');
    require(receipt.timeWithdrawn == 0, 'Can only withdraw once per receipt');

    // close re-entry gate
    receipt.timeWithdrawn = block.timestamp;

    uint[] memory rewards = getRewards(poolId, receiptId);
    pool.totalDepositsWei = pool.totalDepositsWei.minus(receipt.amountDepositedWei);
    bool success = true;

    for (uint i = 0; i < rewards.length; i++) {
        pool.rewardsWeiClaimed[i] = pool.rewardsWeiClaimed[i].plus(rewards[i]);
        success = success && IERC20(pool.rewardTokens[i]).transfer(receipt.owner, rewards[i]);
    }
    success = success && IERC20(pool.depositToken).transfer(receipt.owner, receipt.amountDepositedWei);
    require(success, 'Token transfer failed');

    emit WithdrawalOccurred(poolId, receiptId, receipt.owner);
}
```

Recommendation

Consider introducing an `emergencyExit` function to allow the depositors to withdraw without rewards in case necessary.

4.3 Use `SafeERC20` instead of `IERC20` Medium

Description

Some ERC20 implementations do not implement a return value such as `BNB`. This will cause the token to always revert when trying to transfer tokens.

Examples

code/contracts/MerkleDropFactory.sol:L82

```
token.transferFrom(msg.sender, address(this), value);
```

code/contracts/MerkleDropFactory.sol:L136

```
token.transfer(destination, value);
```

code/contracts/BasicPoolFactory.sol:L136

```
bool success = IERC20(pool.depositToken).transferFrom(msg.sender, address(this), amount);
```

code/contracts/BasicPoolFactory.sol:L159

```
success = success && IERC20(pool.rewardTokens[i]).transfer(receipt.owner, rewards[i]);
```

code/contracts/BasicPoolFactory.sol:L161

```
success = success && IERC20(pool.depositToken).transfer(receipt.owner, receipt.amountDepositedWei);
```

code/contracts/BasicPoolFactory.sol:L177

```
success = success && rewardToken.transfer(management, rewards);
```

code/contracts/BasicPoolFactory.sol:L180

```
success = success && depositToken.transfer(management, depositToken.balanceOf(address(this)));
```

code/contracts/BasicPoolFactory.sol:L136

```
bool success = IERC20(pool.depositToken).transferFrom(msg.sender, address(this), amount);
```

code/contracts/PermissionlessBasicPoolFactory.sol:L472

```
token.transfer(to, amount);
```

code/contracts/PermissionlessBasicPoolFactory.sol:L477

```
token.transferFrom(from, to, amount);
```

Recommendation

It is recommended to always use `safeTransfer/safeTransferFrom` when transferring arbitrary ERC20s. More about on SafeERC20.

4.4 `BasicPoolFactory/PermissionlessBasicPoolFactory.deposit` - Front runners can effectively censor users deposits in some scenarios Minor

Description

The deposit function of both `BasicPoolFactory` and `PermissionlessBasicPoolFactory` is limited for a value defined as `maximumDepositWei`. If the total deposited `wei` is greater than `maximumDepositWei` then the deposit will fail. Front-runners with enough liquidity can spot a `deposit` transaction and “sandwich” it with two transactions, the first is `deposit` and the second is `withdraw` and by doing so the original depositor transaction will fail while the front-runner does not really have a skin in the game in the sense that he does not hold a long term staking position.

Examples

code/contracts/BasicPoolFactory.sol:L118-L140

```
function deposit(uint poolId, uint amount) external {
    Pool storage pool = pools[poolId];
    require(pool.id == poolId, 'Uninitialized pool');
    require(block.timestamp > pool.startTime, 'Cannot deposit before pool start');
    require(block.timestamp < pool.endTime, 'Cannot deposit after pool ends');
    require(pool.totalDepositsWei < pool.maximumDepositWei, 'Maximum deposit already reached');
    if (pool.totalDepositsWei.plus(amount) > pool.maximumDepositWei) {
        amount = pool.maximumDepositWei.minus(pool.totalDepositsWei);
    }
    pool.totalDepositsWei = pool.totalDepositsWei.plus(amount);
    pool.numReceipts = pool.numReceipts.plus(1);

    Receipt storage receipt = pool.receipts[pool.numReceipts];
    receipt.id = pool.numReceipts;
    receipt.amountDepositedWei = amount;
    receipt.timeDeposited = block.timestamp;
    receipt.owner = msg.sender;

    bool success = IERC20(pool.depositToken).transferFrom(msg.sender, address(this), amount);
    require(success, 'Token transfer failed');

    emit DepositOccurred(poolId, pool.numReceipts, msg.sender);
}
```

Recommendation

Consider adding a mechanism to verify that the combination of `deposit-withdraw` is impossible to achieve in a single block.

4.5 A two-step process for changing privileged entities addresses is highly advised Minor

Description

1. `Token.setBank` - `bank` is considered a privileged entity within the `Token` contract, as it is the only account that is allowed to mint additional tokens after the initialization of the contract. `setBank` is used to facilitate the change of the `bank` address, however, in the current implementation in case the wrong address is specified as `newBank`, the ability to mint more tokens by calling `mint` will be lost forever.
2. `BasicPoolFactory.setManagement` - `management` is considered a privileged entity within the `BasicPoolFactory` factory, as it is the only account that is allowed to add pools to the contract. `setManagement` is used to facilitate the change of the `management` address, however, in the current implementation in case the wrong address is specified as `newMgmt`, the ability to add new pools by calling `addPool` will be lost forever.

Examples

code/contracts/Token.sol:L55-L59

```
function setBank(address newBank) public bankOnly {
    address oldBank = bank;
    bank = newBank;
    emit BankUpdated(oldBank, newBank);
}
```

code/contracts/BasicPoolFactory.sol:L59-L64

```
// change the management key
function setManagement(address newMgmt) public managementOnly {
    address oldMgmt = management;
    management = newMgmt;
    emit ManagementUpdated(oldMgmt, newMgmt);
}
```

Recommendation

Consider implementing a two-step process for changing the privileged entities addresses. For more information refer to Ownable2Step.sol

4.6 `Multisend` - Possible missing implementation or wrong inline documentation Minor

Description

The `Multisend.multisend` function is designed to facilitate the distribution of ether to multiple addresses within a single transaction. However, it is important to note that although the inline documentation suggests that the function should reduce network fees, this particular feature is not actually implemented in the code.

Examples

code/contracts/MultiSend.sol:L7-L20

```
// withdrawals enable to multiple withdraws to different accounts
// at one call, and decrease the network fee
function multisend(address payable[] memory addresses, uint256 amount) payable public {
    uint256 total = msg.value;

    for (uint i=0; i < addresses.length; i++) {
        // the total should be greater than the sum of the amounts
        require(total >= amount, "The value is not sufficient");
        total -= amount;

        // send the specified amount to the recipient
        addresses[i].transfer(amount);
    }
}
```

Recommendation

It is recommended to consider implementing the missing network fee reduction feature as described in the inline comment, or alternatively, remove the outdated comment to avoid confusion and provide accurate documentation.

4.7 `Multisend.multisend` - missing input validation on `amount` and `msg.value` Minor

Description

The `Multisend.multisend` function serves the purpose of distributing ether to multiple addresses in a single transaction. Each recipient is allocated an identical amount represented by the variable `amount`. However, it is important to note that in the existing code version, there is a possibility of funds becoming trapped within the contract if the transaction sender inaccurately calculates the total sum intended for transfer in relation to the value of msg.value.

Examples

code/contracts/MultiSend.sol:L9-L21

```
    function multisend(address payable[] memory addresses, uint256 amount) payable public {
        uint256 total = msg.value;

        for (uint i=0; i < addresses.length; i++) {
            // the total should be greater than the sum of the amounts
            require(total >= amount, "The value is not sufficient");
            total -= amount;

            // send the specified amount to the recipient
            addresses[i].transfer(amount);
        }
    }
}
```

Recommendation

Consider adding a validation check to verify that `addresses.length * amount == msg.value`.

Appendix 1 - Files in Scope

This audit covered the following files:

File Name	SHA-1 Hash
contracts/MerkleVesting.sol	bd92bc4bd49549a354c1b7138a170a42b50dff9d
contracts/MerkleDropFactory.sol	7f0b8cc8424fba6dd5c2322e7b203bace2581edd
contracts/TimeVault.sol	09d88585dd015e26bb958d72324e026015b18450
contracts/UniformTimeVault.sol	090336eca84b46ec5e3d115af5c7c161999a6814
contracts/MerkleLib.sol	992bc833537ac194451124ae12a3a1c4210790c3
contracts/DummyMerkleResistor.sol	e9c02631b273cda137efef62c73b80b4e50b8cad
contracts/SafeMathLib.sol	92641094e23458d3e09143e5d9358000cde67a06
contracts/Vault.sol	a46c55c05c9b828e154927160a14d92b9ef1a60e
contracts/MultiSend.sol	c9389b3d4a5c5ec8fc4508926c42adfddac5aa3e
contracts/MerkleResistor.sol	bfb1de63cd8d85a3e844a5a781a79a8e54086dd5
contracts/DummyMerkleDropFactory.sol	29c041cdaa781164711ea61e72fdc52378442c8d
contracts/Token.sol	5bfc99daf528802836e90228adfb84425a46cd76
contracts/DummyMerkleVesting.sol	21806c07aa2e72adca5387af10ddcbb1dadac4e9
contracts/BasicPoolFactory.sol	9eeb5b333122d863692db485f8af4e10517332b8
contracts/PermissionlessBasicPoolFactory.sol	eab86e75880e3cda3e07fd92d9a99dac54b97663

Appendix 2 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.