1inch Liquidity Protocol

Date	December 2020

1 Executive Summary

This report presents the results of our engagement with **1inch Exchange** to provide an initial review of **1inch Liquidity Protocol**.

The review was conducted over two weeks, from **Dec 14 2020** to **Dec 18 2020**. A total of 2x5 (10) person-days were spent.

The review was initially scheduled for a different code-base for a one-week engagement. Due to the time restriction of a single week and the size and complexity of the 1inch Liquidity Protocol code-base, the scope had to be reduced and objectives were prioritized (see section Scope). Given the time constraints, it was agreed to conduct this review on a best-effort basis prioritizing the focus areas.

2 Scope

Our review focused on the commit hash `d3652c992073efed6367ff93f9e8a18dcbd80e9c`. The list of files in scope can be found in the Appendix.

The client requested the following focus areas to be reviewed:

- Changes from Mooniswap-v1 (audit 1, audit 2, audit 3) to the new `1inch Liquidity Protocol`
- The implementation of the new governance system

The following documentation was provided: 1INCH-15499265-101220-1018.pdf

The following files were included in the scope:

```
contracts/Mooniswap.sol 267
contracts/MooniswapConstants.sol 15
contracts/MooniswapDeployer.sol 20
contracts/MooniswapFactory.sol 58
contracts/governance/BaseGovernanceModule.sol 22
contracts/governance/MooniswapFactoryGovernance.sol 143
contracts/governance/MooniswapGovernance.sol 120
contracts/inch/GovernanceMothership.sol 60
contracts/interfaces/IGovernanceModule.sol 5
contracts/interfaces/IMooniswapDeployer.sol 11
contracts/interfaces/IMooniswapFactory.sol 6
contracts/interfaces/IMooniswapFactoryGovernance.sol 11
contracts/interfaces/IReferralFeeReceiver.sol 4
contracts/libraries/LiquidVoting.sol 96
contracts/libraries/Sqrt.sol 18
contracts/libraries/UniERC20.sol 88
contracts/libraries/VirtualBalance.sol 27
contracts/libraries/Vote.sol 34
contracts/utils/BalanceAccounting.sol 21
```

2.1 Scope limitations

- Files not mentioned in the main scope are explicitly excluded from the scope
- Farming contracts were explicitly excluded by the client
- The following contracts were defined as optional scope. Due to time constraints they could not be included.

  ```
  contracts/ReferralFeeReceiver.sol 163
  contracts/governance/GovernanceFeeReceiver.sol 19
  contracts/governance/GovernanceRewards.sol 90
  contracts/utils/Converter.sol 104
  contracts/utils/RewardDistributionRecipient.sol 13
  ```

2.2 Objectives

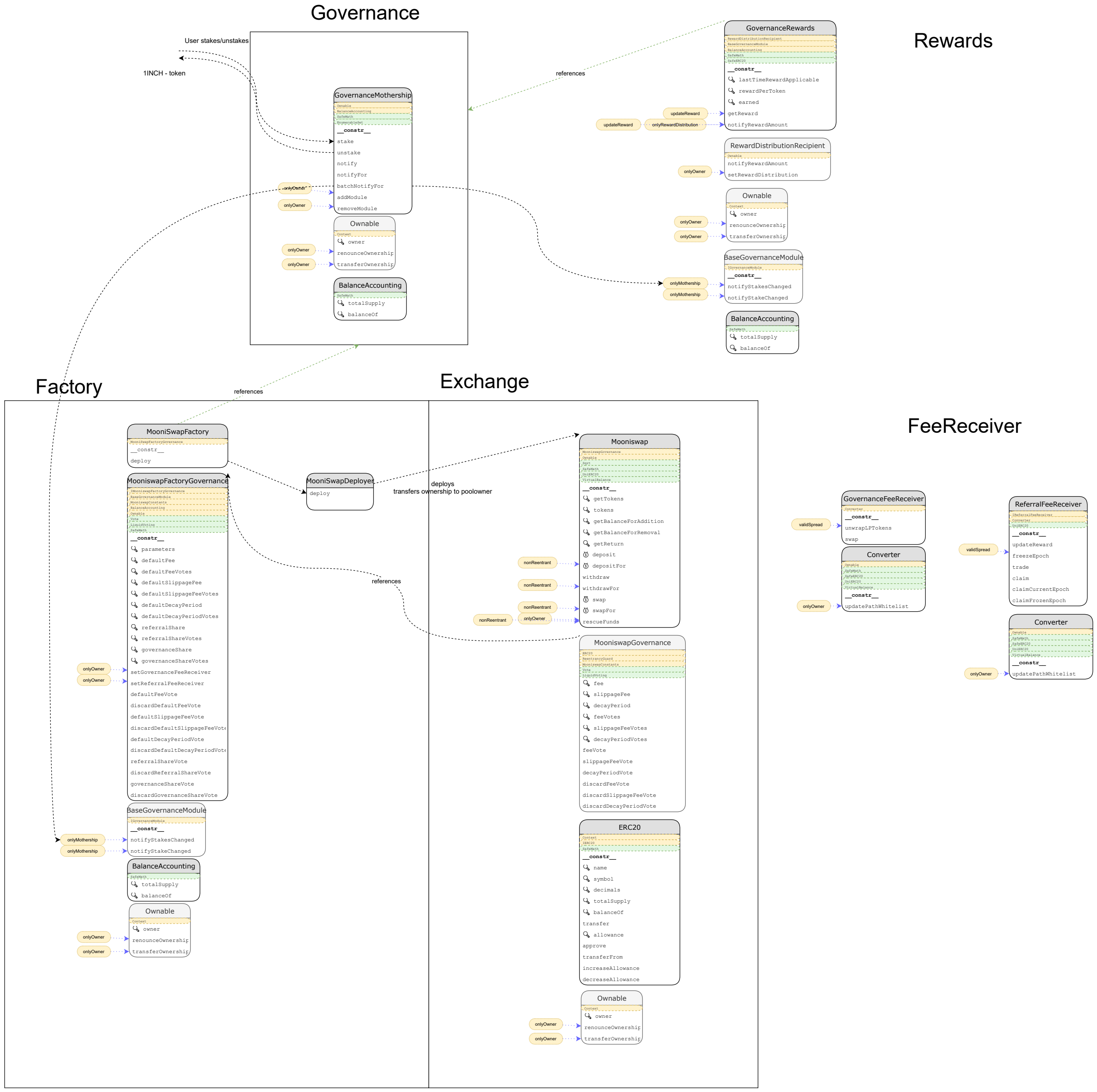
Together with the client team, we identified the following priorities for our review:

1. Ensure that the system is implemented consistently 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. Review changes from `Mooniswap-v1` to `1inch Liquidity Protocol`
4. Review the implementation of the governance system

3 System Overview

The following diagram was created during the audit and helps to illustrate the connections and interdependencies for the contracts in scope.

3.1 Mooniswap



4 Recommendations

4.1 Improve inline documentation and test coverage

Recommendation

The source-units hardly contain any inline documentation which makes it hard to reason about methods and how they are supposed to be used. Consider adding natspec-format compliant inline code documentation, describe functions, what they are used for, and who is supposed to interact with them. Document function or source-unit specific assumptions.

For example, `LiquidVoting` is a concept used to govern system parameters. However, there is no information about how it is specified to work in detail. Furthermore, none of the essential functions of the Mooniswap pool contract have any information about what parameters they expect or potential security risks.

The formal definition of `_getReturn` contains a misprint in `ret = uni_ret * (1 - fee_percentage * slippage)` where `fee_percentage` should be `slip_fee` (see provided pdf document).

code/contracts/Mooniswap.sol:L304-L316

```
/*
    spot_ret = dx * y / x
    uni_ret = dx * y / (x + dx)
    slippage = (spot_ret - uni_ret) / spot_ret
    slippage = dx * dx * y / (x * (x + dx)) / (dx * y / x)
    slippage = dx / (x + dx)
    ret = uni_ret * (1 - fee_percentage * slippage)
    ret = dx * y / (x + dx) * (1 - slip_fee * dx / (x + dx))
    ret = dx * y / (x + dx) * (x + dx - slip_fee * dx) / (x + dx)

    x = amount * denominator
    dx = amount * (denominator - fee)
*/
```

Additionally, test-coverage seems to be limited. Especially for a public-facing exchange contract system test-coverage should be extensive, covering all methods and functions that can directly be accessed including potential security-relevant and edge-cases. This would have helped in detecting some of the findings raised with this report. For example, `Mooniswap.withdrawFor|swapFor` are only covered by `.withdraw|.swap` and never called directly.

4.2 GovernanceMothership - Follow checks-effects-interactions

Description

`Mothership.unstake()` breaks checks-effects-interactions by transferring the requested amount of token out first before validating that the `msg.sender` actually holds the token (with `burn()`). If the `1INCH` token implements callbacks (e.g. `erc777`) this might be more problematic (e.g. allows to flash-lend 1inch token). However, the token is controlled by 1inch and therefore unlikely to pose such a risk.

code/contracts/inch/GovernanceMothership.sol:L36-L42

```
function unstake(uint256 amount) external {
    require(amount > 0, "Empty unstake is not allowed");

    inchToken.transfer(msg.sender, amount);
    _burn(msg.sender, amount);
    _notifyFor(msg.sender, balanceOf(msg.sender));
}
```

Recommendation

`burn()` and `notify()` first, then transfer the token to the user. Consider locking staked token for a minimum amount of `1+` blocks. Consider adding a reentrancy guard. `1INCH` token should not be allowed to have callbacks.

Note that the return value of the transfers is unchecked. As this token is controlled by 1inch and a standard oz `ERC20` we did not consider this as a risk. If you plan on allowing other - potentially spec inconsistent external - tokens as stake consider checking the return values.

4.3 Notes and considerations about the Governance Design

Description

The governance design is based on a master-slave system where users can stake `1INCH` token in the `GovernanceMothership` (the master) which syncs stake to governance sub-modules (e.g. the Factory governance module or the Rewards governance module). An administrative account (or contract or future sub-module) has permissions to add or remove arbitrary sub-modules.

When a user provides stake in the form of `1INCH` token it is automatically synced to sub-modules via the `notify*` family of methods. The same is true when a user requests to unstake their token.

- In a perfect world the public `notify*` family of methods would not be needed as stake is always synced with sub-modules. However, since sub-modules can be added at a later point in time state might not always be in sync, as the newly added module does not trigger state sync.

- Adding a module while the system is already widely accepted requires a manual state-sync by every staker who wants to participate in the sub-modules protocol. This should be documented and it might not be immediately obvious.
- The staking module has its own internal accounting. Voting power is not cleared when a module is removed from the governance mothership. Removing a module disables the stake sync to this module. This might be problematic if a module is first removed, then someone stakes or unstakes, then the module is then re-added. This might leave a user with less voting power or more voting power than staked in the mothership until someone calls `notifyFor` on them.
- Manually updating stake (i.e. by calling `batchNotifyFor`) for all accounts is not feasible. The accounts are not easily accessible (held in a private mapping) and this might easily get very expensive depending on the number of users on the system. According to the client, however, this should not be a problem as users might only propagate their stake when they want to participate in a sub-modules protocol. According to the client, there is no task of keeping the modules 100% in sync.
- The potential to de-sync stake creates an opportunity for an admin to force these kinds of scenarios (see related issues).
- A user is currently allowed to `stake()` and `unstake()` in the same transaction. However, since voting power decays to its maximum only after 24hrs of the vote this is not a big problem.
- Voting dynamics may be unpredictable and favor voting for extremes instead of the actual value someone wants to vote for. For example, one might vote for `0` or `maxValue` to push the weighted average result for the voted value towards higher or lower values. Now if someone removes their stake, this might bounce in a different direction. Ideally, if a lot of accounts honestly vote for the value they want to see, the weighted average should soften this effect but it is unclear if voters will be that rational.
- Different default fees, updated in the factory contract, are still affecting all the existing pools. But the fees of the individual pools are only synced with the factory when the pool stake changes. For example, if only swaps are happening, the default fees values are not updated.

Recommendation

- consider avoiding "double-accounting" of stake and keep it centralized in the mothership to avoid the potential for de-sync's. This would also allow to potentially remove the `notify*` family of methods if `stake()` and `unstake()` is guaranteed to propagate it.
- require a stake to be locked for multiple blocks before it can be unstaked (reject flash-loans)
- require a delay between votes and when they take effect to make the system more predictable
- require a delay when changing administrative settings
- safeguard staking/unstaking with reentrancy guard if the `1INCH` token has callbacks

4.4 Vote - Integer Overflow can be used to force the internal `default` state

Description

The `Vote` data-structure can be forced to represent the `defaultValue` by causing an overflow. Internally, the structure is defined as holding the default value when `Vote.value` is zero. By initializing a `Vote.init(uint_max)` the internal value overflows to zero and the structure will represent the default value.

All occurrences of `Vote.init()` in the current revision of the system are checking at least the upper bounds for the value when setting it. The assessment team did not find an instance where this overflow might become a security risk in the current system. However, this might become a problem in future revisions if the assumption is that a default value can only be created by calling `Vote.init()` without parameters while one can overflow the value to get the same result.

Examples

code/contracts/libraries/Vote.sol:L21-L25

```
function init(uint256 vote) internal pure returns(Vote.Data memory data) {
    return Vote.Data({
        value: vote + 1
    });
}
```

Recommendation

Consider checking for overflows if this might undermine the security of your system.

4.5 Consider using specific contract types in event declarations instead of `address` ✓ Fix Unverified

Resolution

According to the client, this issue is addressed in 1inch-exchange/1inch-liquidity-protocol@`872d97c`

(This fix is as reported by the developer team, but has not been verified by Diligence).

Description

Rather than accepting address parameters and then casting to the known contract type, it is better to use the most specific type possible so the compiler can check for type safety. Typecasting inside the corpus of an event emission is unneeded when the type of the parameter is known beforehand.

Examples

code/contracts/MooniswapFactory.sol:L15-L19

```
event Deployed(
    address indexed mooniswap,
    address indexed token1,
    address indexed token2
);
```

code/contracts/MooniswapFactory.sol:L61-L65

```
emit Deployed(
    address(pool),
    address(token1),
    address(token2)
);
```

Recommendation

Review the complete codebase and, where possible, use more specific types instead of address.

4.6 Unspecific compiler version pragma

Description

For most source-units the compiler version pragma is very unspecific `^0.6.0`. While this often makes sense for libraries to allow them to be included with multiple different versions of an application, it may be a security risk for the actual application implementation itself. A known vulnerable compiler version may accidentally be selected or security tools might fall-back to and older compiler version ending up actually checking a different evm compilation that is ultimately deployed on the blockchain.

Examples

code/contracts/inch/GovernanceMothership.sol:L3-L3

```
pragma solidity ^0.6.0;
```

code/contracts/governance/GovernanceFeeReceiver.sol:L3-L4

```
pragma solidity ^0.6.0;
```

code/contracts/Mooniswap.sol:L2-L4

```
pragma solidity ^0.6.0;
```

and others

Recommendation

Avoid floating pragmas. We highly recommend pinning a concrete compiler version (latest without security issues) in at least the top-level "deployed" contracts to make it unambiguous which compiler version is being used. Rule of thumb: a flattened source-unit should have at least one non-floating concrete solidity compiler version pragma.

4.7 Use of hardcoded gas limits can be problematic

Description

Hardcoded gas limits can be problematic as the past has shown that gas economics in ethereum have changed, and may change again potentially rendering the contract system unusable in the future. Here's a blog post we put out a while ago discussing this topic.

In the specific case in this contract system, gas limits are introduced in `UniERC` to prevent calls to `token.symbol()` to consume too much gas and fail early. The limit may seem reasonable for now but depending on the expected lifetime of the contract may get problematic when gas economics change in the future. A similar problem exists with `address.transfer()` calls that receive a gas stipend. Be conscious about this potential limitation and prepare for the case where gas prices might change in a way that negatively affects the contract system.

5 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.

5.1 [Out of Scope] ReferralFeeReceiver - anyone can steal all the funds that belong to ReferralFeeReceiver Critical

✓ Fix Unverified

Resolution

According to the client, this issue is addressed in 1inch-exchange/1inch-liquidity-protocol#2 and the reentrancy in `FeeReceiver` in 1inch-exchange/1inch-liquidity-protocol@ `e9c6a03`

(This fix is as reported by the developer team, but has not been verified by Diligence).

Description

Note: *This issue was raised in components that were being affected by the scope reduction as outlined in the section "Scope" and are, therefore, only shallowly validated. Nevertheless, we find it important to communicate such potential findings and ask the client to further investigate.*

The `ReferralFeeReceiver` receives pool shares when users `swap()` tokens in the pool. A `ReferralFeeReceiver` may be used with multiple pools and, therefore, be a lucrative target as it is holding pool shares.

Any token or `ETH` that belongs to the `ReferralFeeReceiver` is at risk and can be drained by any user by providing a custom `mooniswap` pool contract that references existing token holdings.

It should be noted that none of the functions in `ReferralFeeReceiver` verify that the user-provided `mooniswap` pool address was actually deployed by the linked `MooniswapFactory`. The factory provides certain security guarantees about mooniswap pool contracts (e.g. valid mooniswap contract, token deduplication, tokenA!=tokenB, enforced token sorting, …), however, since the `ReferralFeeReceiver` does not verify the user-provided `mooniswap` address they are left unchecked.

Additional Notes

- `freezeEpoch` - (callable by anyone) performs a `pool.withdraw()` with the `minAmounts` check being disabled. This may allow someone to call this function at a time where the contract actually gets a bad deal.
- `trade` - (callable by anyone) can intentionally be used to perform bad trades (front-runnable)
- `trade` - (callable by anyone) appears to implement inconsistent behavior when sending out `availableBalance`. `ETH` is sent to `tx.origin` (the caller) while tokens are sent to the user-provided `mooniswap` address.

code/contracts/ReferralFeeReceiver.sol:L91-L95

```
if (path[0].isETH()) {
    tx.origin.transfer(availableBalance);  // solhint-disable-line avoid-tx-o
} else {
    path[0].safeTransfer(address(mooniswap), availableBalance);
}
```

- multiple methods - since `mooniswap` is a user-provided address there are a lot of opportunities to reenter the contract. Consider adding reentrancy guards as another security layer (e.g. `claimCurrentEpoch` and others).
- multiple methods - do not validate the amount of tokens that are returned, causing an evm assertion due to out of bounds index access.

code/contracts/ReferralFeeReceiver.sol:L57-L59

```
IERC20[] memory tokens = mooniswap.getTokens();
uint256 token0Balance = tokens[0].uniBalanceOf(address(this));
uint256 token1Balance = tokens[1].uniBalanceOf(address(this));
```

- in `GovernanceFeeReceiver` anyone can intentionally force unwrapping of pool tokens or perform swaps in the worst time possible. e.g. The checks for `withdraw(..., minAmounts)` is disabled.

code/contracts/governance/GovernanceFeeReceiver.sol:L18-L26

```
function unwrapLPTokens(Mooniswap mooniswap) external validSpread(mooniswap)
    mooniswap.withdraw(mooniswap.balanceOf(address(this)), new uint256[](0))
}

function swap(IERC20[] memory path) external validPath(path) {
    (uint256 amount,) = _maxAmountForSwap(path, path[0].uniBalanceOf(address
    uint256 result = _swap(path, amount, payable(address(rewards)));
    rewards.notifyRewardAmount(result);
}
```

Examples

A malicious user can drain all token by calling `claimFrozenEpoch` with a custom contract as `mooniswap` that returns a token address the `ReferralFeeReceiver` contracts holds token from in `IERC20[] memory tokens = mooniswap.getTokens();`. A subsequent call to `_transferTokenShare()` will then send out any amount of token requested by the attacker to the attacker-controlled address (`msg.sender`).

Let's assume the following scenario:

- `ReferralFeeReceiver` holds `DAI` token and we want to steal them.

An attacker may be able to drain the contract from `DAI` token via `claimFrozenToken` if

- they control the `mooniswap` address argument and provide a malicious contract
- `user.share[mooniswap][firstUnprocessedEpoch] > 0` - this can be arbitrarily set in `updateReward`
- `token.epochBalance[currentEpoch].token0Balance > 0` - this can be manipulated in `freezeEpoch` by providing a malicious `mooniswap` contract
- they own a worthless `ERC20` token e.g. named `ATTK`

The following steps outline the attack:

1. The attacker calls into `updateReward` to set `user.share[mooniswap][currentEpoch]` to a value that is greater than zero to make sure that `share` in `claimFrozenEpoch` takes the `_transferTokenShare` path.

code/contracts/ReferralFeeReceiver.sol:L38-L50

```
function updateReward(address referral, uint256 amount) external override {
    Mooniswap mooniswap = Mooniswap(msg.sender);
    TokenInfo storage token = tokenInfo[mooniswap];
    UserInfo storage user = userInfo[referral];
    uint256 currentEpoch = token.currentEpoch;

    // Add new reward to current epoch
    user.share[mooniswap][currentEpoch] = user.share[mooniswap][currentEpoch
    token.epochBalance[currentEpoch].totalSupply = token.epochBalance[currer

    // Collect all processed epochs and advance user token epoch
    _collectProcessedEpochs(user, token, mooniswap, currentEpoch);
}
```

1. The attacker then calls `freezeEpoch()` providing the malicious `mooniswap` contract address controlled by the attacker.
   - The malicious contract returns token that is controlled by the attacker (e.g. `ATTK`) in a call to `mooniswap.getTokens();`
   - The contract then stores the current balance of the attacker-controlled token in `token0Balance/token1Balance`. Note that the token being returned here by the malicious contract can be different from the one we're checking out in the last step (balance manipulation via `ATTK`, checkout of `DAI` in the last step).
   - Then the contract calls out to the malicious `mooniswap` contract. This gives the malicious contract an easy opportunity to send some attacker-controlled token (`ATTK`) to the `ReferralFeeReceiver` in order to freely manipulate the frozen tokenbalances (`tokens[0].uniBalanceOf(address(this)).sub(token0Balance);`).
   - Note that the used token addresses are never stored anywhere. The balances recorded here are for an attacker-controlled token (`ATTK`), not the actual one that we're about to steal (e.g. `DAI`)
   - The token balances are now set-up for checkout in the last step (`claimFrozenEpoch`).

code/contracts/ReferralFeeReceiver.sol:L52-L64

```
function freezeEpoch(Mooniswap mooniswap) external validSpread(mooniswap) {
    TokenInfo storage token = tokenInfo[mooniswap];
    uint256 currentEpoch = token.currentEpoch;
    require(token.firstUnprocessedEpoch == currentEpoch, "Previous epoch is

    IERC20[] memory tokens = mooniswap.getTokens();
    uint256 token0Balance = tokens[0].uniBalanceOf(address(this));
    uint256 token1Balance = tokens[1].uniBalanceOf(address(this));
    mooniswap.withdraw(mooniswap.balanceOf(address(this)), new uint256[](0))
    token.epochBalance[currentEpoch].token0Balance = tokens[0].uniBalanceOf(
    token.epochBalance[currentEpoch].token1Balance = tokens[1].uniBalanceOf(
    token.currentEpoch = currentEpoch.add(1);
}
```

1. A call to `claimFrozenEpoch` checks-out the previously frozen token balance.
   - The `claim > 0` requirement was fulfilled in step 1.
   - The token balance was prepared for the attacker-controlled token (`ATTK`) in step 2, but we're now checking out `DAI` .
   - When the contract calls out to the attackers `mooniswap` contract the call to `IERC20[] memory tokens = mooniswap.getTokens();` returns the address of the token to be stolen (e.g. `DAI`) instead of the attacker-controlled token (`ATTK`) that was used to set-up the balance records.
   - Subsequently, the valuable target tokens (`DAI`) are sent out to the caller in `_transferTokenShare` .

code/contracts/ReferralFeeReceiver.sol:L153-L162

```
if (share > 0) {
    EpochBalance storage epochBalance = token.epochBalance[firstUnprocessedE
    uint256 totalSupply = epochBalance.totalSupply;
    user.share[mooniswap][firstUnprocessedEpoch] = 0;
    epochBalance.totalSupply = totalSupply.sub(share);

    IERC20[] memory tokens = mooniswap.getTokens();
    epochBalance.token0Balance = _transferTokenShare(tokens[0], epochBalance
    epochBalance.token1Balance = _transferTokenShare(tokens[1], epochBalance
    epochBalance.inchBalance = _transferTokenShare(inchToken, epochBalance.i
```

Recommendation

Enforce that the user-provided `mooniswap` contract was actually deployed by the linked factory. Other contracts cannot be trusted. Consider implementing token sorting and de-duplication (tokenA!=tokenB) in the pool contract constructor as well. Consider employing a reentrancy guard to safeguard the contract from reentrancy attacks.

Improve testing. The methods mentioned here are not covered at all. Improve documentation and provide a specification that outlines how this contract is supposed to be used.

Review the “additional notes” provided with this issue.

5.2 GovernanceMothership - `notifyFor` allows to arbitrarily create new or override other users stake in governance modules Critical ✓ Fix Unverified

Resolution

According to the client, this issue is addressed in 1inch-exchange/1inch-liquidity-protocol@ `2ce549d` and added tests with 1inch-exchange/1inch-liquidity-protocol@ `e0dc46b`

(This fix is as reported by the developer team, but has not been verified by Diligence).

Description

The `notify*` methods are called to update linked governance modules when an accounts stake changes in the Mothership. The linked modules then update their own balances of the user to accurately reflect the account’s real stake in the Mothership.

Besides `notify` there’s also a method named `notifyFor` which is publicly accessible. It is assumed that the method should be used similar to `notify` to force an update for another account’s balance.

However, invoking the method forces an update in the linked modules for the provided address, **but takes** `balanceOf(msg.sender)` instead of `balanceOf(account)`. This allows malicious actors to:

- Arbitrarily change other accounts stake in linked governance modules (e.g. zeroing stake, increasing stake) based on the callers stake in the mothership
- Duplicate stake out of thin air to arbitrary addresses (e.g. staking in mothership once and calling `notifyFor` many other account addresses)

Examples

- publicly accessible method allows forcing stake updates for arbitrary users

code/contracts/inch/GovernanceMothership.sol:L48-L50

```
function notifyFor(address account) external {
    _notifyFor(account, balanceOf(msg.sender));
}
```

- the method calls the linked governance modules

code/contracts/inch/GovernanceMothership.sol:L73-L78

```
function _notifyFor(address account, uint256 balance) private {
    uint256 modulesLength = _modules.length();
    for (uint256 i = 0; i < modulesLength; ++i) {
        IGovernanceModule(_modules.at(i)).notifyStakeChanged(account, balanc
    }
}
```

- which will arbitrarily `mint` or `burn` stake in the `BalanceAccounting` of `Factory` or `Reward` (or other linked governance modules)

code/contracts/governance/BaseGovernanceModule.sol:L29-L31

```
function notifyStakeChanged(address account, uint256 newBalance) external ov
    _notifyStakeChanged(account, newBalance);
}
```

code/contracts/governance/MooniswapFactoryGovernance.sol:L144-L160

```
function _notifyStakeChanged(address account, uint256 newBalance) internal c
    uint256 balance = balanceOf(account);
    if (newBalance > balance) {
        _mint(account, newBalance.sub(balance));
    } else if (newBalance < balance) {
        _burn(account, balance.sub(newBalance));
    } else {
        return;
    }
    uint256 newTotalSupply = totalSupply();

    _defaultFee.updateBalance(account, _defaultFee.votes[account], balance,
    _defaultSlippageFee.updateBalance(account, _defaultSlippageFee.votes[acc
    _defaultDecayPeriod.updateBalance(account, _defaultDecayPeriod.votes[acc
    _referralShare.updateBalance(account, _referralShare.votes[account], bal
    _governanceShare.updateBalance(account, _governanceShare.votes[account],
}
```

code/contracts/governance/GovernanceRewards.sol:L72-L79

```
function _notifyStakeChanged(address account, uint256 newBalance) internal c
    uint256 balance = balanceOf(account);
    if (newBalance > balance) {
        _mint(account, newBalance.sub(balance));
    } else if (newBalance < balance) {
        _burn(account, balance.sub(newBalance));
    }
}
```

Recommendation

Remove `notifyFor` or change it to take the balance of the correct account `_notifyFor(account, balanceOf(msg.sender))`.

It is questionable whether the public `notify*()` family of methods is actually needed as stake should only change - and thus an update of linked modules should only be required - if an account calls `stake()` or `unstake()`. It should therefore be considered to remove `notify()`, `notifyFor` and `batchNotifyFor`.

5.3 Users can "increase" their voting power by voting for the max/min values Medium

Description

Many parameters in the system are determined by the complicated governance mechanism. These parameters are calculated as a result of the voting process and are equal to the weighted average of all the votes that stakeholders make. The idea is that every user is voting for the desired value. But if the result value is smaller (larger) than the desired, the user can change the vote for the max (min) possible value. That would shift the result towards the desired one and basically “increase” this stakeholder’s voting power. So every user is more incentivized to vote for the min/max value than for the desired one.

The issue’s severity is not high because all parameters have reasonable max value limitations, so it’s hard to manipulate the system too much.

Recommendation

Reconsider the voting mechanism.

5.4 The `uniTransferFrom` function can potentially be used with invalid params Medium ✓ Fix Unverified

Resolution

According to the client, this issue is addressed in 1inch-exchange/1inch-liquidity-protocol@`d0ffb6f`.

(This fix is as reported by the developer team, but has not been verified by Diligence).

Description

The system is using the `UniERC20` contract to incapsulate transfers of both ERC-20 tokens and ETH. This contract has `uniTransferFrom` function that can be used for any ERC-20 or ETH:

code/contracts/libraries/UniERC20.sol:L36-L48

```
function uniTransferFrom(IERC20 token, address payable from, address to, uir
    if (amount > 0) {
        if (isETH(token)) {
            require(msg.value >= amount, "UniERC20: not enough value");
            if (msg.value > amount) {
                // Return remainder if exist
                from.transfer(msg.value.sub(amount));
            }
        } else {
            token.safeTransferFrom(from, to, amount);
        }
    }
}
```

In case if the function is called for the normal ERC-20 token, everything works as expected. The tokens are transferred from the `from` address to the `to` address. If the token is ETH - the transfer is expected to be from the `msg.sender` to `this` contract. Even if the `to` and `from` parameters are different.

This issue's severity is not high because the function is always called with the proper parameters in the current codebase.

Recommendation

Make sure that the `uniTransferFrom` function is always called with expected parameters.

5.5 MooniswapGovernance - votingpower is not accurately reflected when minting pool tokens Medium ✓ Fix Unverified

Resolution

According to the client, this issue is addressed in 1inch-exchange/1inch-liquidity-protocol@`eb869fd`

(This fix is as reported by the developer team, but has not been verified by Diligence).

Description

When a user provides liquidity to the pool, pool-tokens are minted. The minting event triggers the `_beforeTokenTransfer` callback in `MooniswapGovernance` which updates voting power reflecting the newly minted stake for the user.

There seems to be a copy-paste error in the way `balanceTo` is determined that sets `balanceTo` to zero if new token were minted (`from==address(0)`). This means, that in a later call to `_updateOnTransfer` only the newly minted amount is considered when adjusting voting power.

Examples

- If tokens are newly minted `from==address(0)` and therefore `balanceTo -> 0`.

code/contracts/governance/MooniswapGovernance.sol:L100-L114

```
function _beforeTokenTransfer(address from, address to, uint256 amount) inte
    uint256 balanceFrom = (from != address(0)) ? balanceOf(from) : 0;
    uint256 balanceTo = (from != address(0)) ? balanceOf(to) : 0;
    uint256 newTotalSupply = totalSupply()
        .add(from == address(0) ? amount : 0)
        .sub(to == address(0) ? amount : 0);

    ParamsHelper memory params = ParamsHelper({
        from: from,
        to: to,
        amount: amount,
        balanceFrom: balanceFrom,
        balanceTo: balanceTo,
        newTotalSupply: newTotalSupply
    });
```

- now, `balanceTo` is zero which would adjust voting power to `amount` instead of the user's actual balance + the newly minted token.

code/contracts/governance/MooniswapGovernance.sol:L150-L153

```
if (params.to != address(0)) {
    votingData.updateBalance(params.to, voteTo, params.balanceTo, params.bal
}
```

Recommendation

`balanceTo` should be zero when burning (`to == address(0)`) and `balanceOf(to)` when minting.

e.g. like this:

```
uint256 balanceTo = (to != address(0)) ? balanceOf(to) : 0;
```

5.6 MooniswapGovernance - `_beforeTokenTransfer` should not update voting power on transfers to self Medium

✓ Fix Unverified

Resolution

Addressed 1inch-exchange/1inch-liquidity-protocol@ `7c7126d`

(This fix is as reported by the developer team, but has not been verified by Diligence).

Description

Mooniswap governance is based on the liquidity voting system that is also employed by the mothership or for factory governance. In contrast to traditional voting systems where users vote for discrete values, the liquidity voting system derives a continuous weighted averaged “consensus” value from all the votes. Thus it is required that whenever stake changes in the system, all the parameters that can be voted upon are updated with the new weights for a specific user.

The Mooniswap pool is governed by liquidity providers and liquidity tokens are the stake that gives voting rights in `MooniswapGovernance` . Thus whenever liquidity tokens are transferred to another address, stake and voting values need to be updated. This is handled by `MooniswapGovernance._beforeTokenTransfer()` .

In the special case where someone triggers a token transfer where the `from` address equals the `to` address, effectively sending the token to themselves, no update on voting power should be performed. Instead, voting power is first updated with `balance - amount` and then with `balance + amount` which in the worst case means it is updating first to a zero balance and then to 2x the balance.

Ultimately this should not have an effect on the overall outcome but is unnecessary and wasting gas.

Examples

- `beforeTokenTransfer` callback in `Mooniswap` does not check for the NOP case where `from==to`

code/contracts/governance/MooniswapGovernance.sol:L100-L119

```
function _beforeTokenTransfer(address from, address to, uint256 amount) inte
    uint256 balanceFrom = (from != address(0)) ? balanceOf(from) : 0;
    uint256 balanceTo = (from != address(0)) ? balanceOf(to) : 0;
    uint256 newTotalSupply = totalSupply()
        .add(from == address(0) ? amount : 0)
        .sub(to == address(0) ? amount : 0);

    ParamsHelper memory params = ParamsHelper({
        from: from,
        to: to,
        amount: amount,
        balanceFrom: balanceFrom,
        balanceTo: balanceTo,
        newTotalSupply: newTotalSupply
    });

    _updateOnTransfer(params, mooniswapFactoryGovernance.defaultFee, _emitFe
    _updateOnTransfer(params, mooniswapFactoryGovernance.defaultSlippageFee,
    _updateOnTransfer(params, mooniswapFactoryGovernance.defaultDecayPeriod,
}
```

- which leads to `updateBalance` being called on the same address twice, first with `currentBalance - amountTransferred` and then with `currentBalance + amountTransferred`.

code/contracts/governance/MooniswapGovernance.sol:L147-L153

```
if (params.from != address(0)) {
    votingData.updateBalance(params.from, voteFrom, params.balanceFrom, para
}

if (params.to != address(0)) {
    votingData.updateBalance(params.to, voteTo, params.balanceTo, params.bal
}
```

Recommendation

Do not update voting power on LP token transfers where `from == to`.

5.7 Unpredictable behavior for users due to admin front running or general bad timing Medium

Description

In a number of cases, administrators of contracts can update or upgrade things in the system without warning. This has the potential to violate a security goal of the system.

Specifically, privileged roles could use front running to make malicious changes just ahead of incoming transactions, or purely accidental negative effects could occur due to the unfortunate timing of changes.

In general users of the system should have assurances about the behavior of the action they're about to take.

Examples

MooniswapFactoryGovernance - Admin opportunity to lock `swapFor` with a referral when setting an invalid `referralFeeReceiver`

- `setReferralFeeReceiver` and `setGovernanceFeeReceiver` takes effect immediately.

code/contracts/governance/MooniswapFactoryGovernance.sol:L92-L95

```
function setReferralFeeReceiver(address newReferralFeeReceiver) external onl
    referralFeeReceiver = newReferralFeeReceiver;
    emit ReferralFeeReceiverUpdate(newReferralFeeReceiver);
}
```

- `setReferralFeeReceiver` can be used to set an invalid receiver address (or one that reverts on every call) effectively rendering `Mooniswap.swapFor` unusable if a referral was specified in the swap.

code/contracts/Mooniswap.sol:L281-L286

```
if (referral != address(0)) {
    referralShare = invIncrease.mul(referralShare).div(_FEE_DENOMINATOR);
    if (referralShare > 0) {
        if (referralFeeReceiver != address(0)) {
            _mint(referralFeeReceiver, referralShare);
            IReferralFeeReceiver(referralFeeReceiver).updateReward(referral,
```

Locking staked token

At any point in time and without prior notice to users an admin may accidentally or intentionally add a broken governance sub-module to the system that blocks all users from unstaking their `1INCH` token. An admin can recover from this by removing the broken sub-module, however, with malicious intent tokens may be locked forever.

Since `1INCH` token gives voting power in the system, tokens are considered to hold value for other users and may be traded on exchanges. This raises concerns if tokens can be locked in a contract by one actor.

- An admin adds an invalid address or a malicious sub-module to the governance contract that always `reverts` on calls to `notifyStakeChanged`.

code/contracts/inch/GovernanceMothership.sol:L63-L66

```
function addModule(address module) external onlyOwner {
    require(_modules.add(module), "Module already registered");
    emit AddModule(module);
}
```

code/contracts/inch/GovernanceMothership.sol:L73-L78

```
function _notifyFor(address account, uint256 balance) private {
    uint256 modulesLength = _modules.length();
    for (uint256 i = 0; i < modulesLength; ++i) {
        IGovernanceModule(_modules.at(i)).notifyStakeChanged(account, balanc
    }
}
```

Admin front-running to prevent user stake sync

An admin may front-run users while staking in an attempt to prevent submodules from being notified of the stake update. This is unlikely to happen as it incurs costs for the attacker (front-back-running) to normal users but may be an interesting attack scenario to exclude a whale's stake from voting.

For example, an admin may front-run `stake()` or `notoify*()` by briefly removing all governance submodules from the mothership and re-adding them after the users call succeeded. The stake-update will not be propagated to the sub-modules. A user may only detect this when they are voting (if they had no stake before) or when they actually check their stake. Such an attack might likely stay unnoticed unless someone listens for `addmodule` `removemodule` events on the contract.

- An admin front-runs a transaction by removing all modules and re-adding them afterwards to prevent the stake from propagating to the submodules.

code/contracts/inch/GovernanceMothership.sol:L68-L71

```
function removeModule(address module) external onlyOwner {
    require(_modules.remove(module), "Module was not registered");
    emit RemoveModule(module);
}
```

Admin front-running to prevent unstake from propagating

An admin may choose to front-run their own `unstake()`, temporarily removing all governance sub-modules, preventing `unstake()` from syncing the action to sub-modules while still getting their previously staked tokens out. The governance sub-modules can be re-added right after unstaking. Due to double-accounting of the stake (in governance and in every sub-module) their stake will still be exercisable in the sub-module even though it was removed from the mothership. Users can only prevent this by manually calling a state-sync on the affected account(s).

Recommendation

The underlying issue is that users of the system can't be sure what the behavior of a function call will be, and this is because the behavior can change at any time.

We recommend giving the user advance notice of changes with a time lock. For example, make all system-parameter and upgrades require two steps with a mandatory time window between them. The first step merely broadcasts to users that a particular change is coming, and the second step commits that change after a suitable waiting period. This allows users that do not accept the change to withdraw immediately.

Furthermore, users should be guaranteed to be able to redeem their staked tokens. An entity - even though trusted - in the system should not be able to lock tokens indefinitely.

5.8 The owner can borrow token0/token1 in the `rescueFunds` Minor

Description

If some random tokens/funds are accidentally transferred to the pool, the `owner` can call the `rescueFunds` function to withdraw any funds manually:

code/contracts/Mooniswap.sol:L331-L340

```
function rescueFunds(IERC20 token, uint256 amount) external nonReentrant onl
    uint256 balance0 = token0.uniBalanceOf(address(this));
    uint256 balance1 = token1.uniBalanceOf(address(this));

    token.uniTransfer(msg.sender, amount);

    require(token0.uniBalanceOf(address(this)) >= balance0, "Mooniswap: acce
    require(token1.uniBalanceOf(address(this)) >= balance1, "Mooniswap: acce
    require(balanceOf(address(this)) >= _BASE_SUPPLY, "Mooniswap: access der
}
```

There's no restriction on which funds the `owner` can try to withdraw and which token to call. It's theoretically possible to transfer pool tokens and then return them to the contract (e.g. in the case of ERC-777). That action would be similar to a free flash loan.

Recommendation

Explicitly check that the `token` is not equal to any of the pool tokens.

Appendix 1 - Files in Scope

This audit covered the following files:

Main Focus

File Name	SHA-1 Hash
contracts/Mooniswap.sol	659eff36b8eaa58fef019769c11721c9f3ad0189
contracts/MooniswapConstants.sol	a789d0aff24373a8d9a310787c098e2b2aa9a36e
contracts/MooniswapDeployer.sol	5737ffee01d702c1d998d0322bc100e17b97f982
contracts/MooniswapFactory.sol	d2905081ed86eb643d09d452f97cd10b0a11bc0c
contracts/governance/BaseGovernanceModule.sol	ec4063cebc143cf899f5f81b8f25041f8bcbbbe0
contracts/governance/MooniswapFactoryGovernance.sol	57e67f788e67fed1fc327594a9d6703d34a729dd
contracts/governance/MooniswapGovernance.sol	3c924bd39426854a7d5f294ae15448c2081f8d72
contracts/inch/GovernanceMothership.sol	4e1cf98b5c8a246f4ebf662311e9392a6c1d9626
contracts/interfaces/IGovernanceModule.sol	c36dc05ec446b75eae41b18fb93fc38d61105b4d

File Name	SHA-1 Hash
contracts/interfaces/IMooniswapDeployer.sol	1850983116a4f24e88dec9e1273dffcb81fc9f33
contracts/interfaces/IMooniswapFactory.sol	8705f3e3b2c502744f202d09882ec2b352833aac
contracts/interfaces/IMooniswapFactoryGovernance.sol	a2b5b2af928f8bb2fddcef789f216f52436e7574
contracts/interfaces/IReferralFeeReceiver.sol	06ff903c88095b2530a67c75207d0356e3a57033
contracts/libraries/LiquidVoting.sol	012ac90e3a5388e651030d850c5245357f8c970e
contracts/libraries/Sqrt.sol	5835dd72ef2e6356da1d5d723aeb548e397b1248
contracts/libraries/UniERC20.sol	e0aa3564b8c4c2f8d24d4efcf8b88f090feab98d
contracts/libraries/VirtualBalance.sol	2b5fc9544e12034611cab31178c7019917f6c91d
contracts/libraries/Vote.sol	00600853f6eaaed3c31abf8c3643bf0c5077d965
contracts/utils/BalanceAccounting.sol	753e87aa6e67cb747678cbed8692a30532e1a47e

Optional Scope

File Name	SHA-1 Hash
contracts/ReferralFeeReceiver.sol	5e9e95520e2d5274e132df981064a1b8be1ee4a7
contracts/governance/GovernanceFeeReceiver.sol	6a602e40d86e6c6b9484b4c5aa605d8f7ce6b10e
contracts/governance/GovernanceRewards.sol	364eb7e2107e0b7ddabaf2db7aecc81b55349629

File Name	SHA-1 Hash
contracts/utils/Converter.sol	0cfce090515be2e3e32440010838ca5d8f91e31c
contracts/utils/RewardDistributionRecipient.sol	32defc0f715f11f9cd83aa63601255ea14c9107e

Explicitly out of scope

File Name	SHA-1 Hash
contracts/libraries/Voting.sol	bb1ea2150e48a81bbe147c2f98f8dfb67c87851c
contracts/inch/farming/FarmingRewards.sol	0660032fcea9ba91c0fa8ec77c370739254039ce
contracts/inch/farming/FarmingVoter.sol	221cc83637519bea61ba88361a23e91c2285a433

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 Solidity code and only the Solidity code we note as being within the scope of our review within this report. 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 Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

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.

6 Document Change Log

Version	Date	Description
1.0	2020-12-21	Report Delivery
1.1	2020-12-22	Updated Recommendations 4.1 and 4.3 and removed invalid statements. Updated 5.1 to better describe the issue
1.2	2021-02-18	Renamed project from **mooniswap-v2** to **1inch Liquidity Protocol** as per clients request. Updated issues with feedback/fixes the client provided.