GrowthDeFi WHEAT

Date	June 2021
Lead Auditor	Sergii Kravchenko
Co-auditors	David Oz Kashi, Dominik Muhs

1 Executive Summary

This report presents the results of our engagement with **GrowthDeFi** to review **the WHEAT protocol**.

The review was conducted over three weeks, from **June 14, 2021** to **July 2, 2021** by **Sergii Kravchenko, David Oz Kashi**, and **Dominik Muhs**. A total of 30 person-days were spent.

During the first week, we familiarized ourselves with the GrowthDeFi system, economic incentives in it, and peripheral infrastructure such as PantherSwap, PancakeSwap, and AutoFarm.

During the second week we inspected the strategy token contracts, their interactions with third-party infrastructure, and their effect on the rest of the system. Furthermore, the distribution of rewards and potentially vulnerable user flows have been analyzed.

During the third week we focused on the fees collectors and inspected peripheral infrastructure a bit deeper.

Scope

Our review focused on the `audit` branch, specifically, commit hash `8360ac0a537589bb974e8a5a169bb3e7c95d2857`. The list of files in scope can be found in the Appendix.

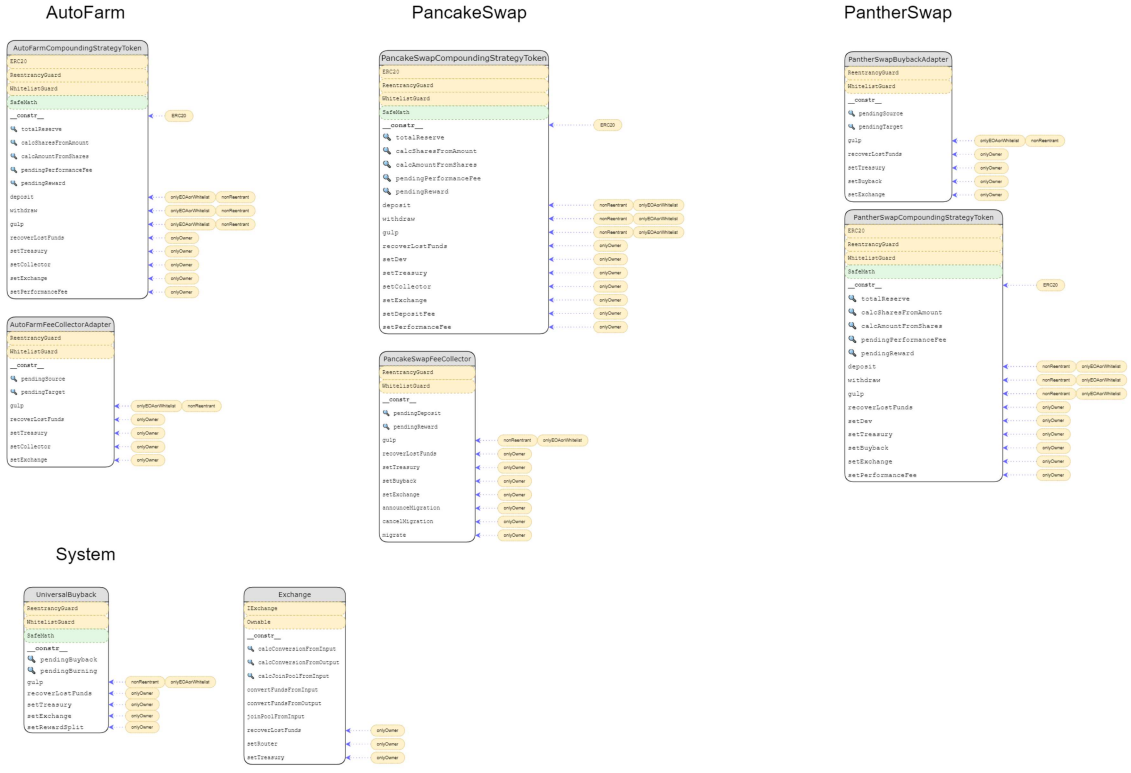
2.1 Objectives

Together with the **GrowthDeFi** 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.

A specific focus has been put on preventing attacks that can result in the loss of system/user funds.

3 System Overview



4 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

4.1 Actors

The relevant actors are listed below with their respective abilities:

- User/EOA
- Peripheral Project
- Strategy Token
- Fee Collector
- Buyback Actor

4.2 Trust Assumptions

The system's security heavily relies on the security of the integrated third parties. It is assumed that exchanges and tokens exhibit correct and consistent behaviour, e.g. in terms of token balances and transfers. Therefore, we strongly recommend the implementation of a thorough vetting process before a token is integrated into the GrowthDeFi ecosystem. A checklist for an integration candidate could include points such as:

- Reviewing audit report(s) performed by well-known, independent professionals,
- Reviewing the adherence to standard interfaces (ERC-20/BEP-20, MasterChef, Uniswap token pair),
- Ensuring that security-critical components are not upgradable,
- Ensuring that the candidate's development team has a process in place to handle security incidents,
- Ensuring that a point of contact is known to handle potential legal issues.

Furthermore, trust is placed on the functionality of `tx.origin`, which may be deprecated in the future. More details are outlines in the issues section.

5 Recommendations

5.1 Test suite improvements

Description

The test suite at this stage is not complete. For complicated systems such as GrowthDefi, which uses many different modules and interacts with different DeFi protocols, it is crucial to have a full test coverage (could be verified with `solidity-coverage` tool) that includes both unit tests and integration tests to cover both the correctness of the core logic and possible interface issues. As we've seen in some smart contract incidents, a complete test suite can prevent issues that might be hard to find with manual reviews.

5.2 Refactoring of duplicate code regions

Description

Across the code base, especially in the strategy token contracts, there are highly similar code regions. Duplication of such code may become an issue in future development as fixes may be applied only partially by accident, resulting in inconsistent system behaviour and hard-to-find bugs.

Examples

In the strategy token contracts:

- `calcSharesFromAmount` and other view functions
- `deposit` and `withdraw` checks and control flow
- `gulp` conversions from reward- to reserve-tokens
- `recoverLostFunds` business logic

Recommendation

It is recommended to deduplicate the codebase by inheriting generic contracts that provide common control flow and business logic actions. Then, by overriding local methods, implementing the concrete strategy token behaviour specific to each third-party interaction.

This will reduce the potential for future bugs and increase the codebase's maintainability and extensibility.

6 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.

6.1 Frontrunning attacks by the `owner` Major

Resolution
The client communicated this issue was addressed in commit 34c6b355795027d27ae6add7360e61eb6b01b91b.

Description

There are few possible attack vectors by the owner:

1. All strategies have fees from rewards. In addition to that, the PancakeSwap strategy has deposit fees. The default deposit fees equal zero; the maximum is limited to 5%:

   wheat-v1-core-audit/contracts/PancakeSwapCompoundingStrategyToken.sol:L29-L33

```
uint256 constant MAXIMUM_DEPOSIT_FEE = 5e16; // 5%
uint256 constant DEFAULT_DEPOSIT_FEE = 0e16; // 0%

uint256 constant MAXIMUM_PERFORMANCE_FEE = 50e16; // 50%
uint256 constant DEFAULT_PERFORMANCE_FEE = 10e16; // 10%
```

When a user deposits tokens, expecting to have zero deposit fees, the `owner` can frontrun the deposit and increase fees to 5%. If the deposit size is big enough, that may be a significant amount of money.

2. In the `gulp` function, the reward tokens are exchanged for the reserve tokens on the `exchange`:

wheat-v1-core-audit/contracts/PancakeSwapCompoundingStrategyToken.sol:L218-L244

```
function gulp(uint256 _minRewardAmount) external onlyEOAorWhitelist nonl
{
  uint256 _pendingReward = _getPendingReward();
  if (_pendingReward > 0) {
      _withdraw(0);
  }
  {
      uint256 _totalReward = Transfers._getBalance(rewardToken);
      uint256 _feeReward = _totalReward.mul(performanceFee) / 1e18;
      Transfers._pushFunds(rewardToken, collector, _feeReward);
  }
  if (rewardToken != routingToken) {
      require(exchange != address(0), "exchange not set");
      uint256 _totalReward = Transfers._getBalance(rewardToken);
      Transfers._approveFunds(rewardToken, exchange, _totalReward);
      IExchange(exchange).convertFundsFromInput(rewardToken, routingTok
  }
  if (routingToken != reserveToken) {
      require(exchange != address(0), "exchange not set");
      uint256 _totalRouting = Transfers._getBalance(routingToken);
      Transfers._approveFunds(routingToken, exchange, _totalRouting);
      IExchange(exchange).joinPoolFromInput(reserveToken, routingToken,
  }
  uint256 _totalBalance = Transfers._getBalance(reserveToken);
  require(_totalBalance >= _minRewardAmount, "high slippage");
  _deposit(_totalBalance);
}
```

The `owner` can change the `exchange` parameter to the malicious address that steals tokens. The `owner` then calls `gulp` with `_minRewardAmount==0`, and all the rewards will be stolen. The same attack can be implemented in fee collectors and the buyback contract.

Recommendation

Use a timelock to avoid instant changes of the parameters.

6.2 New deposits are instantly getting a share of undistributed rewards Major

Resolution
The client communicated this issue was addressed in commit 34c6b355795027d27ae6add7360e61eb6b01b91b.

Description

When a new deposit is happening, the current pending rewards are not withdrawn and re-invested yet. And they are not taken into account when calculating the number of shares that the depositor receives. The number of shares is calculated as if there were no pending rewards. The other side of this issue is that all the withdrawals are also happening without considering the pending rewards. So currently, it makes more sense to withdraw right after `gulp` to gather the rewards. In addition to the general "unfairness" of the reward distribution during the deposit/withdrawal, there is also an attack vector created by this issue.

The Attack

If the deposit is made right before the `gulp` function is called, the rewards from the `gulp` are distributed evenly across all the current deposits, including the ones that were just recently made. So if the deposit-gulp-withdraw sequence is executed, the caller receives guaranteed profit. If the attacker also can execute these functions briefly (in one block or transaction) and take a huge loan to deposit a lot of tokens, almost all the rewards from the gulp will be stolen by the attacker. The easy 1-transaction attack with a flashloan can be done by the owner, miner, whitelisted contracts, or any contract if the `onlyEOAorWhitelist` modifier is disabled or stops working (https://github.com/ConsenSys/growthdefi-audit-2021-06/issues/3). Even if `EOAorWhitelist` is working properly, anyone can take a regular loan to make

the attack. The risk is not that big because no price manipulation is required. The price will likely remain the same during the attack (few blocks maximum).

Recommendation

If issue issue 6.3 is fixed while allowing anyone call the gulp contract, the best solution would be to include the `gulp` call at the beginning of the `deposit` and `withdraw`. In case of withdrawing, there should also be an option to avoid calling `gulp` as the emergency case.

6.3 Proactive sandwiching of the `gulp` calls Major

Resolution
The client communicated this issue was addressed in commit 34c6b355795027d27ae6add7360e61eb6b01b91b.

Description

Each strategy token contract provides a `gulp` method to fetch pending rewards, convert them into the reserve token and split up the balances. One share is sent to the fee collector as a performance fee, while the rest is deposited into the respective `MasterChef` contract to accumulate more rewards. Suboptimal trades are prevented by passing a minimum slippage value with the function call, which results in revert if the expected reserve token amount cannot be provided by the trade(s).

The slippage parameter and the trades performed in `gulp` open the function up to proactive sandwich attacks. The slippage parameter can be freely set by the attacker, resulting in the system performing arbitrarily bad trades based on how much the attacker can manipulate the liquidity of involved assets around the `gulp` function call.

This attack vector is significant under the following assumptions:

- The exchange the trade is performed on allows significant changes in liquidity pools in a single transaction (e.g., not limiting transactions to X% of the pool amount),

- The attacker can frontrun legitimate `gulp` calls with reasonable slippage values,
- Trades are performed, i.e. when `rewardToken != routingToken` and/or `routingToken != reserveToken` hold true.

Examples

This affects the `gulp` functions in all the strategies:

- `PancakeSwapCompoundingStrategyToken`
- `AutoFarmCompoundingStrategyToken`
- `PantherSwapCompoundingStrategyToken`

and also fees collectors and the buyback adapters:

- `PantherSwapBuybackAdapter`
- `AutoFarmFeeCollectorAdapter`
- `PancakeSwapFeeCollector`
- `UniversalBuyback`

Recommendation

There are different possible solutions to this issue and all have some tradeoffs. Initially, we came up with the following suggestion:

- The `onlyOwner` modifier should be added to the `gulp` function to ensure only authorized parties with reasonable slippages can execute trades on behalf of the strategy contracts. Furthermore, additional slippage checks can be added to avoid unwanted behavior of authorized addresses, e.g., to avoid a bot setting unreasonable slippage values due to a software bug.

But in order to fix another issue (https://github.com/ConsenSys/growthdefi-audit-2021-06/issues/8), we came up with the alternative solution:

- Use oracles to restrict users from calling the `gulp` function with unreasonable slippage (more than 5% from the oracle's moving average price). The side effect of that solution is that sometimes the outdated price will be used. That means that when the price crashes, nobody will be able to call the `gulp`.

6.4 Expected amounts of tokens in the `withdraw` function

Medium

Resolution

Client's statement : "This issue did not really need fixing. The mitigation was already in place by depositing a tiny amount of the reserve into the contract, if necessary"

Description

Every `withdraw` function in the strategy contracts is calculating the expected amount of the returned tokens before withdrawing them:

wheat-v1-core-audit/contracts/PantherSwapCompoundingStrategyToken.sol:L200-L208

```
function withdraw(uint256 _shares, uint256 _minAmount) external onlyEOAorWhi
{
        address _from = msg.sender;
        (uint256 _amount, uint256 _withdrawalAmount, uint256 _netAmount) = _
        require(_netAmount >= _minAmount, "high slippage");
        _burn(_from, _shares);
        _withdraw(_amount);
        Transfers._pushFunds(reserveToken, _from, _withdrawalAmount);
}
```

After that, the contract is trying to transfer this pre-calculated amount to the `msg.sender`. It is never checked whether the intended amount was actually transferred to the strategy contract. If the amount is lower, that may result in reverting the `withdraw` function all the time and locking up tokens.

Even though we did not find any specific case of returning a different amount of tokens, it is still a good idea to handle this situation to minimize relying on the security of the external contracts.

Recommendation

There are a few options how to mitigate the issue:

- Double-check the balance difference before and after the MasterChef's `withdraw` function is called.
- Handle this situation in the emergency mode (https://github.com/ConsenSys/growthdefi-audit-2021-06/issues/11).

6.5 Emergency mode of the MasterChef contracts is not supported Medium

Resolution

The client communicated this issue was addressed in commit 34c6b355795027d27ae6add7360e61eb6b01b91b.

Description

All the underlying MasterChef contracts have the emergency withdrawal mode, which allows simpler withdrawal (excluding the rewards):

```
// Withdraw without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw(uint256 _pid) public nonReentrant {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    uint256 amount = user.amount;
    user.amount = 0;
    user.rewardDebt = 0;
    user.rewardLockedUp = 0;
    user.nextHarvestUntil = 0;
    pool.lpToken.safeTransfer(address(msg.sender), amount);
    emit EmergencyWithdraw(msg.sender, _pid, amount);
}
```

```
    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
        emit EmergencyWithdraw(msg.sender, _pid, user.amount);
        user.amount = 0;
        user.rewardDebt = 0;
    }
```

```
    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public nonReentrant {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        uint256 wantLockedTotal =
            IStrategy(poolInfo[_pid].strat).wantLockedTotal();
        uint256 sharesTotal = IStrategy(poolInfo[_pid].strat).sharesTotal();
        uint256 amount = user.shares.mul(wantLockedTotal).div(sharesTotal);

        IStrategy(poolInfo[_pid].strat).withdraw(msg.sender, amount);

        pool.want.safeTransfer(address(msg.sender), amount);
        emit EmergencyWithdraw(msg.sender, _pid, amount);
        user.shares = 0;
        user.rewardDebt = 0;
    }
```

While it's hard to predict how and why the emergency mode can be enabled in the underlying MasterChef contracts, these functions are there for a reason, and it's safer to be able to use them. If some emergency happens and this is the only way to withdraw funds, the funds in the strategy contracts will be locked forever.

Recommendation

Add the emergency mode implementation.

6.6 The capping mechanism for Panther token leads to increased fees Medium

Resolution

The client communicated this issue was addressed in commit 34c6b355795027d27ae6add7360e61eb6b01b91b.

Description

Panther token has a cap in transfer sizes, so any transfer in the contract is limited beforehand:

wheat-v1-core-audit/contracts/PantherSwapCompoundingStrategyToken.sol:L218-L245

```
function gulp(uint256 _minRewardAmount) external onlyEOAorWhitelist nonReent
{
        uint256 _pendingReward = _getPendingReward();
        if (_pendingReward > 0) {
                _withdraw(0);
        }
        uint256 __totalReward = Transfers._getBalance(rewardToken);
        (uint256 _feeReward, uint256 _retainedReward) = _capFeeAmount(__tota
        Transfers._pushFunds(rewardToken, buyback, _feeReward);
        if (rewardToken != routingToken) {
                require(exchange != address(0), "exchange not set");
                uint256 _totalReward = Transfers._getBalance(rewardToken);
                _totalReward = _capTransferAmount(rewardToken, _totalReward,
                Transfers._approveFunds(rewardToken, exchange, _totalReward)
                IExchange(exchange).convertFundsFromInput(rewardToken, routi
        }
        if (routingToken != reserveToken) {
                require(exchange != address(0), "exchange not set");
                uint256 _totalRouting = Transfers._getBalance(routingToken);
                _totalRouting = _capTransferAmount(routingToken, _totalRouti
                Transfers._approveFunds(routingToken, exchange, _totalRoutir
                IExchange(exchange).joinPoolFromInput(reserveToken, routingT
        }
        uint256 _totalBalance = Transfers._getBalance(reserveToken);
        _totalBalance = _capTransferAmount(reserveToken, _totalBalance, _ret
        require(_totalBalance >= _minRewardAmount, "high slippage");
        _deposit(_totalBalance);
}
```

Fees here are calculated from the full amount of rewards (`__totalReward`):

wheat-v1-core-audit/contracts/PantherSwapCompoundingStrategyToken.sol:L225

```
(uint256 _feeReward, uint256 _retainedReward) = _capFeeAmount(__totalReward.
```

But in fact, if the amount of the rewards is too big, it will be capped, and the residuals will be "taxed" again during the next call of the `gulp` function. That behavior leads to multiple taxations of the same tokens, which means increased fees.

Recommendation

The best solution would be to cap `__totalReward` first and then calculate fees from the capped value.

6.7 The `_capFeeAmount` function is not working as intended Medium

Resolution
Client's statement : "With the fix of 6.6 this code was removed and therefore no changes were required. "

Description

Panther token has a limit on the transfer size. Because of that, all the Panther transfer values in the `PantherSwapCompoundingStrategyToken` are also capped beforehand. The following function is called to cap the size of fees:

wheat-v1-core-audit/contracts/PantherSwapCompoundingStrategyToken.sol:L357-L366

```
function _capFeeAmount(uint256 _amount) internal view returns (uint256 _capp
{
        _retained = 0;
        uint256 _limit = _calcMaxRewardTransferAmount();
        if (_amount > _limit) {
                _amount = _limit;
                _retained = _amount.sub(_limit);
        }
        return (_amount, _retained);
}
```

This function should return the capped amount and the amount of retained tokens. But because the `_amount` is changed before calculating the `_retained`, the retained amount will always be 0.

Recommendation

Calculate the `retained` value before changing the `amount`.

6.8 Stale split ratios in `UniversalBuyback` Medium

Resolution
The client communicated this issue was addressed in commit 34c6b355795027d27ae6add7360e61eb6b01b91b.

Description

The `gulp` and `pendingBurning` functions of the `UniversalBuyback` contract use the hardcoded, constant values of `DEFAULT_REWARD_BUYBACK1_SHARE` and `DEFAULT_REWARD_BUYBACK2_SHARE` to determine the ratio the trade value is split with.

Consequently, any call to `setRewardSplit` to set a new ratio will be ineffective but still result in a `ChangeRewardSplit` event being emitted. This event can deceive system operators and users as it does not reflect the correct values of the contract.

Examples

wheat-v1-core-audit/contracts/UniversalBuyback.sol:L80-L81

```
uint256 _amount1 = _balance.mul(DEFAULT_REWARD_BUYBACK1_SHARE) / 1e18;
uint256 _amount2 = _balance.mul(DEFAULT_REWARD_BUYBACK2_SHARE) / 1e18;
```

wheat-v1-core-audit/contracts/UniversalBuyback.sol:L97-L98

```
uint256 _amount1 = _balance.mul(DEFAULT_REWARD_BUYBACK1_SHARE) / 1e18;
uint256 _amount2 = _balance.mul(DEFAULT_REWARD_BUYBACK2_SHARE) / 1e18;
```

Recommendation

Instead of the default values, `rewardBuyback1Share` and `rewardBuyback2Share` should be used.

6.9 Future-proofness of the `onlyEOAorWhitelist` modifier Medium

Resolution
The client communicated this issue was addressed in commit 34c6b355795027d27ae6add7360e61eb6b01b91b.

Description

The `onlyEOAorWhitelist` modifier is used in various locations throughout the code. It performs a check that asserts the message sender being equal to the transaction origin to assert the calling party is not a smart contract.

This approach may stop working if EIP-3074 and its AUTH and AUTHCALL opcodes get deployed.

While the OpenZeppelin reentrancy guard does not depend on `tx.origin`, the EOA check does. Its evasion can result in additional attack vectors such as flash loans opening up. It is noteworthy that preventing smart contract interaction with the protocol may limit its opportunities as smart contracts

cannot integrate with it in the same way that GrowthDeFi integrates with its third-party service providers.

The `onlyEOAorWhitelist` modifier may give a false sense of security because it won't allow making a flash loan attack by most of the users. But the same attack can still be made by some people or with more risk:

- The owner and the whitelisted contracts are not affected by the modifier.
- The modifier can be disabled:

```
**wheat-v1-core-audit/contracts/WhitelistGuard.sol:L21-L28**
```solidity
modifier onlyEOAorWhitelist()
{
  if (enabled) {
      address _from = _msgSender();
      require(tx.origin == _from || whitelist.contains(_from), "access denied");
  }
  _;
}
```
```

  And in the deployment script, this modifier is disabled for testing purposes, and it's important not to forget to turn it in on the production:

  wheat-v1-core-audit/migrations/02_deploy_contracts.js:L50

```
  await pancakeSwapFeeCollector.setWhitelistEnabled(false); // allows test
```

- The attack can usually be split into multiple transactions. Miners can put these transactions closely together and don't take any additional risk. Regular users can take a risk, take the loan, and execute the attack in multiple transactions or even blocks.

Recommendation

It is strongly recommended to monitor the progress of this EIP and its potential implementation on the Binance Smart Chain. If this functionality gets enabled, the development team should update the contract system to use the new opcodes. We also strongly recommend relying less on the fact that only EOA will call the functions. It is better to write the code that can be used by the external smart contracts without compromising its security.

6.10 Exchange owner might steal users' funds using reentrancy Medium

Resolution

The client communicated this issue was addressed in commit 34c6b355795027d27ae6add7360e61eb6b01b91b.

Description

The practice of pulling funds from a user (by using `safeTransferFrom`) and then later pushing (some) of the funds back to the user occurs in various places in the `Exchange` contract. In case one of the used token contracts (or one of its dependent calls) externally calls the `Exchange` owner, the owner may utilize that to call back `Exchange.recoverLostFunds` and drain (some) user funds.

Examples

wheat-v1-core-audit/contracts/Exchange.sol:L80-L89

```
function convertFundsFromInput(address _from, address _to, uint256 _inputAmo
{
        address _sender = msg.sender;
        Transfers._pullFunds(_from, _sender, _inputAmount);
        _inputAmount = Math._min(_inputAmount, Transfers._getBalance(_from))
        _outputAmount = UniswapV2ExchangeAbstraction._convertFundsFromInput(
        _outputAmount = Math._min(_outputAmount, Transfers._getBalance(_to))
        Transfers._pushFunds(_to, _sender, _outputAmount);
        return _outputAmount;
}
```

wheat-v1-core-audit/contracts/Exchange.sol:L121-L130

```
function joinPoolFromInput(address _pool, address _token, uint256 _inputAmou
{
        address _sender = msg.sender;
        Transfers._pullFunds(_token, _sender, _inputAmount);
        _inputAmount = Math._min(_inputAmount, Transfers._getBalance(_token)
        _outputShares = UniswapV2LiquidityPoolAbstraction._joinPoolFromInput
        _outputShares = Math._min(_outputShares, Transfers._getBalance(_pool
        Transfers._pushFunds(_pool, _sender, _outputShares);
        return _outputShares;
}
```

wheat-v1-core-audit/contracts/Exchange.sol:L99-L111

```
function convertFundsFromOutput(address _from, address _to, uint256 _outputA
{
        address _sender = msg.sender;
        Transfers._pullFunds(_from, _sender, _maxInputAmount);
        _maxInputAmount = Math._min(_maxInputAmount, Transfers._getBalance(_
        _inputAmount = UniswapV2ExchangeAbstraction._convertFundsFromOutput(
        uint256 _refundAmount = _maxInputAmount - _inputAmount;
        _refundAmount = Math._min(_refundAmount, Transfers._getBalance(_from
        Transfers._pushFunds(_from, _sender, _refundAmount);
        _outputAmount = Math._min(_outputAmount, Transfers._getBalance(_to))
        Transfers._pushFunds(_to, _sender, _outputAmount);
        return _inputAmount;
}
```

wheat-v1-core-audit/contracts/Exchange.sol:L139-L143

```
function recoverLostFunds(address _token) external onlyOwner
{
        uint256 _balance = Transfers._getBalance(_token);
        Transfers._pushFunds(_token, treasury, _balance);
}
```

Recommendation

Reentrancy guard protection should be added to `Exchange.convertFundsFromInput`, `Exchange.convertFundsFromOutput`, `Exchange.joinPoolFromInput`, `Exchange.recoverLostFunds` at least, and in general to all public/external functions since gas price considerations are less relevant for contracts deployed on BSC.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
./AutoFarmCompoundingStrategyToken.sol	c682e1f7d6d0acfd26933ad3169dbbd2fdd4562c
./AutoFarmFeeCollectorAdapter.sol	41961d71d902acc31dcfd1274afa5a7f060ae671
./Exchange.sol	5d83f2881d5e3e2053fa96b97fcd53595e1192dc
./IExchange.sol	35e7cff12a28758d502ff452b512ef99959150e2
./interop/AutoFarmV2.sol	f2fd89fc8cfac68225af9cd47c36e5080255238a
./interop/Belt.sol	3ff6a4bcbe7eb59de56f144f476aa66337d69715
./interop/MasterChef.sol	7547c901e7068cf19dff4e8265dfb2669f06143d
./interop/PantherSwap.sol	83401998c19ae3c47fa01d61a6ea1a22c394a655
./interop/UniswapV2.sol	929d36dd4ec3b53364423b54098a076b5fefd85b
./interop/WrappedToken.sol	f735d7d325ac4b20e5447a532aced5d4f7c31b8a
./Migrations.sol	55bfb09493c7ecea45ed2ab9366db665af70aee2
./modules/Math.sol	2fcff034aba0c7dec9b7f5caae6295b21372871c
./modules/Transfers.sol	a7439175b42844b3b8ff7b593987a33fd6ceb3ee

File	SHA-1 hash
./modules/UniswapV2ExchangeAbstraction.sol	cf58e61b9a4583cbb57fe34ef7a54b2f5237033f
./modules/UniswapV2LiquidityPoolAbstraction.sol	f8e0e3dd5de61da29871b249f701279cbb23304c
./modules/Wrapping.sol	e02e0c9380dc3a281a5d6f43c0a9e5e39d854764
./network/$.sol	1d15880a1c99ba39e26020d40144bf7325f0b642
./PancakeSwapCompoundingStrategyToken.sol	b8841d52f589292bd5b5759977917a8846bcbad8
./PancakeSwapFeeCollector.sol	c303eaf8ff61b5c4dd8f45b2f7fa417422c790a9
./PantherSwapBuybackAdapter.sol	13449e0644b560640089c560aff4954ffd5177bf
./PantherSwapCompoundingStrategyToken.sol	e687c9f356c91a3933bbc4a3c74a065c57ed156e
./UniversalBuyback.sol	209a21322c45c9e1da92503d69fb794530a2dcd3
./WhitelistGuard.sol	d6fb7ddca4a85222e58196693306c75fbd777f42

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.

POLICY