Trader Joe

3 March 2022

by Ackee Blockchain

Contents

1. Document Revisions

1.0	Final report	March 9, 2022

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specialized in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run a free certification course Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, Rockaway Blockchain Fund.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Slither is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzzy testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzzy tests.

2.3. Review team

Member’s Name	Position
Dominik Teiml	Lead Auditor
Jan Kalivoda	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.4. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Trader Joe is a defi monolotith based on Avalanche. It allows users to trade, lend, and stake assets on Avalanche.

Between Feb 7 and March 4, 2022, Trader Joe engaged ABCH to conduct a security review of several new components in their system. We reviewed the following contracts:

1. MoneyMaker
2. StableJoeStaking
3. VeJoeToken
4. VeJoeStaking
5. BoostedMasterChefJoe

Note that the commit for the first two was 210af8bf5d, while it was 9ae7edc7a7 for the latter three. All five contracts were brand new, they were not upgrades to existing contracts. In the period mentioned above, we were allocated three engineering weeks and the lead auditor was Dominik Teiml.

We began our review by using static analysis tools, namely Slither and the solc compiler. This yielded several issues such as H3: Many components lack data validation. We then took a deep dive into the logic of the contracts. During the review, we paid special attention to:

- Is the correctness of the two contracts ensured?
- Do the contracts correctly use dependencies or other contracts they rely on, namely JoePair?
- Are access controls not too relaxed or too strict?
- Are the upgradeable contracts subject to common upgradeability pitfalls?

- Is the code vulnerable to re-entrancy attacks, either through ERC777-style contracts, or maliciously supplied user input?

Finally, we also fuzzed BoostedMasterChefJoe. We supplied the fuzzing model to the Client.

Our review resulted in 18 findings, ranging from Informational to High severity. The most critical was that a denial of service could occur in BoostedMasterChefJoe under relatively common circumstances (see H1: `BoostedMasterChefJoe` may get stuck due to an invariant violation). We recommend taking a deep look at the arithmetic to ensure it is correct, and heavily testing the resultant code with the fuzzing model.

Ackee Blockchain recommends Trader Joe:

- heavily test BoostedMasterChefJoe with our fuzzing model,
- address all reported issues,
- build on top of the fuzzing model during future development and use it to test the safety and correctness of any future code.

Finally, it should be noted that the Client has chosen to remain pseudonymous.

4. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandibility purposes and does not constitute a formal specification.

4.1. Contracts

MoneyMaker

The MoneyMaker receives 5 b.p. (0.05%) of every swap done on Trader Joe in an LP token. It unwraps this LP coin to obtain the underlying two tokens, and then uses a system of bridges to swap both underlying coins to a specific `tokenTo` (designated to be a stablecoin such as USDC). It then sends that to StableJoeStaking.

StableJoeStaking

StableJoeStaking allows users to stake Joe token to receive rewards. These accrue to users in a fashion proportional to the total amount of Joe staked in the contract at the time the rewards are received (more specifically, at the time `updateRewards` is called).

BoostedMasterChefJoe

BoostedMasterChefJoe is similar to MasterChefJoe with the exception that User's veJoe tokens provide users a boost in Joe rewards. Since MasterChefJoeV2 is currently the only contract with Joe minting rights, it is implemented that BoostedMasterChefJoe is a staker in MasterChefJoeV2. MasterChefJoeV2 mints it Joe, and it then distributes that to its own stakers.

VeJoeToken

Inherits from `VeERC20`, which is a non-transferrable, ERC20-like token.

VeJoeToken has an Ownable pattern that is used for minting and burning.

VeJoeStaking

Users stake JOE into veJOE and will accrue veJOE over time. veJOE provides a Farm Boost to JOE rewards on select farms.

The contract has the following parameters (in its `initialize` function):

- `_joe` - Address of the JOE token contract
- `_veJoe` - Address of the veJOE token contract
- `_veJoePerSharePerSec` - veJOE per sec per JOE staked, scaled to `VEJOE_PER_SHARE_PER_SEC_PRECISION`
- `_speedUpVeJoePerSharePerSec` - Similar to `_veJoePerSharePerSec` but for speed up
- `_speedUpThreshold` - Percentage of total staked JOE user has to deposit receive speed up
- `_speedUpDuration` - Length of time a user receives speed up benefits
- `_maxCap` - The maximum amount of veJOE received per JOE staked

The rate of staking can be speeded up, when the deposit amount is higher or equal to the actual staked amount multiplied by `_speedUpThreshold`. The speeding up is also enabled when it is the first deposit.

Unstaking **any** amount of JOE means the user will lose all of his current veJOE tokens.

4.2. Actors

This part describes actors of the system, their roles and permissions.

MoneyMaker

Owner

The MoneyMaker has an `owner`, by default the deployer. The owner may transfer its ownership through `Ownable.transferOwnership`.

The `owner`, and only the `owner`, may set the dev cut, dev address, token to address.

He can also add and remove (see I1: `MoneyMaker.authorized` keeps old values) users from the `auth` list, or make it permissionless.

Allowlist

Users on the `auth` list may set bridges for tokens (see M1: Setting `anyAuth` to `true` leads to undefined behavior), and call convert, which performs the unwrapping, swapping for reward token, as well as sending to StableJoeStaking

StableJoeStaking

Owner

StableJoeStaking also has an `owner`, by default the deployer. The owner may transfer ownership through `OwnableUpgradeable.transferOwnership`.

The `owner`, and only the `owner`, may add and remove reward tokens and set the deposit fee percent.

VeJoeToken

Owner

The owner of VeJoeToken. Token Owner is able to mint or burn tokens, transfer or renounce his ownership. For purposes of VeJoeStaking it should be set to the address of VeJoeStaking contract.

VeJoeStaking

Owner

The owner of VeJoeStaking. Staking Owner is able to change parameters listed in VeJoeStaking except the token addresses. `_maxCap` can be set only to a higher value than was set on `initialize`. Furthermore, he can transfer or renounce his ownership.

BoostedMasterChefJoe

Owner

The `owner` is set to the initializer by default. The are able to add new supported LP tokens and update their allocation points and transfer and renounce ownership.

4.3. Trust Model

Users have to trust:

- MoneyMaker owner since he can change the token to parameter, resulting in impossibility to conver tokens,
- StableJoeStaking owner since they can change the deposit fee percent up to 50%,
- VeJoeStaking owner since he can change the conditions of staking inconveniently (from the user's perspective),
- VeJoeToken owner since he can change the behavior of `mint` and `burn` arbitrarily,
- BoostedMasterChefJoe owner since he can diminish an LP token's allocation points on demand.

5. Vulnerabilities risk methodology

Each finding contains an `Impact` and `Likelihood` ratings.

If we have found a scenario in which the issue is exploitable, it will be assigned an impact of `Critical`, `High`, `Medium`, or `Low`, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of `Warning` or `Informational`.

`Low` to `Critical` impact issues also have a `Likelihood` which measures the probability of exploitability during runtime.

5.1. Finding classification

The full definitions are as follows:

Impact

High

Code that activates the issue will lead to undefined or catastrophic consequences for the system.

Medium

Code that activates the issue will result in consequences of serious substance.

Low

Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.

Warning

The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as "Warning" or higher, based on our best estimate of whether it is currently exploitable.

Informational

The issue is on the border-line between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

High

The issue is exploitable by virtually anyone under virtually any circumstance.

Medium

Exploiting the issue currently requires non-trivial preconditions.

Low

Exploiting the issue requires strict preconditions.

6. Findings

This section contains the list of discovered findings. Unless overriden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*, and
- a *Recommendation*

Many times, there might be multiple ways to solve or alleviate the issue, with varying requirements in terms of the necessary changes to the codebase. In that case, we will try to enumerate them all, making clear which solve the underlying issue better (albeit possibly only with architectural changes) than others.

Summary of Findings

	Type	Impact	Likelihood
H1: `BoostedMasterChefJoe` may get stuck due to an invariant violation	Denial of service	High	High
H2: Transferring tokens to `BoostedMasterChefJoe` before first deposit may cause DoS	Denial of service	High	Medium
H3: Many components lack data validation	Data validation	High	Low
H4: Renounce ownership	Access Control	High	Low
H5: `setBoostedMasterChefJoe` has insufficient data validation	Data validation	High	Low

	Type	Impact	Likelihood
H6: Tokens with callbacks	Token interaction, Re-entrancy	High	Low
H7: Usage of `solc` optimizer	Compiler configuration	High	Low
M1: Setting `anyAuth` to `true` leads to undefined behavior	Access controls	Medium	High
M3: Renewing boosting period can fail	Logic fault	Medium	High
M2: Array lengths are not validated	Data validation	Medium	Medium
L1: Code duplication in `MoneyMaker`	Code duplication	Low	N/A
W1: OpenZeppelin dependencies contain bugs	Dependencies	Warning	N/A
W2: Front-runnning initialize function	Front-running	Warning	N/A
W3: `getPendingVeJoe` does not have up to date values	Function behaviour	Warning	N/A
W4: Pre-0.8 `solc` versions don’t check for arithmetic overflow	Compiler configuration	Warning	N/A
I1: `MoneyMaker.authorized` keeps old values	Data consistency	Informational	High
I2: Use `_msgSender` over `msg.sender`	Builtin variables	Informational	N/A

	Type	Impact	Likelihood
I3: Log old values in logs	Logging	Informatio nal	High

Table 1. Table of Findings

H1: `BoostedMasterChefJoe` may get stuck due to an invariant violation

Impact:	High	Likelihood:	High
Target:	BoostedMasterChefJoe	Type:	Denial of service

Description

Many functions, including `withdraw`, `deposit` and `pendingTokens`, have the following expression:

Listing 1. BoostedMasterChefJoe.sol#L361-L365

```
        uint256 pending = boostedLiquidity
            .mul(pool.accJoePerShare)
            .div(ACC_TOKEN_PRECISION)
            .sub(user.rewardDebt)
            .add(claimableJoe[_pid][msg.sender]);
```

During our testing, there were many situations where this expression reverted at the subtraction step. This would cause a denial of service, making it impossible to deposit or withdraw for the user (until `boostedLiquidity` is large enough).

Exploit scenario

Given the fuzzing setup we supplied to the Client, here is a sequence that triggers this bug:

```
lps[2].approve(s.bmcj, 1e18, {'from': alice})
bmcj.deposit(2, 1e18, {'from': alice})

# advance time by 1 hour
chain.sleep(60 * 60)
chain.mine()

vejoe.mint(alice, 1e18)

vejoe.mint(bob, 1e18)

lps[2].approve(s.bmcj, 0, {'from': bob})
bmcj.deposit(2, 0, {'from': bob})

# advance time by 1 hour
chain.sleep(60 * 60)
chain.mine()

bmcj.deposit(2, 0, {'from': alice})
```

Recommendation

Short term, ensure that the above script passes by correcting the corresponding arithmetic issue.

Long term, build on top of the fuzzing model to detect issues such as this during testing.

Go back to Findings Summary

H2: Transferring tokens to `BoostedMasterChefJoe` before first deposit may cause DoS

Impact:	High	Likelihood:	Medium
Target:	BoostedMasterChefJoe	Type:	Denial of service

Listing 2. BoostedMasterChefJoe.sol#L296-L298

```
    function deposit(uint256 _pid, uint256 _amount) external nonReentrant {
        harvestFromMasterChef();
        updatePool(_pid);
```

Listing 3. BoostedMasterChefJoe.sol#L276-L283

```
    function updatePool(uint256 _pid) public {
        PoolInfo memory pool = poolInfo[_pid];
        if (block.timestamp > pool.lastRewardTimestamp) {
            uint256 lpSupply = pool.lpToken.balanceOf(address(this));
            if (lpSupply != 0) {
                uint256 secondsElapsed = block.timestamp.sub(pool.lastRewardTimestamp);
                uint256 joeReward = secondsElapsed.mul(joePerSec()).mul(pool.allocPoint).div(totalAllocPoint);
                pool.accJoePerShare = pool.accJoePerShare.add(
```

Description

When `BoostedMasterChefJoe.deposit` is called, `BoostedMasterChefJoe.updatePool` is called (see Listing 2). If time has elapsed since the last time pool rewards were updated and the contract's lp token balance is non-zero, the contract updates `pool.accJoePerShare`. In that assignment expression, the contract divides by `pool.totalBoostedAmount` (see Listing 3).

The issue is that it is possible for the preconditions to be true, yet `pool.totalBoostedAmount` be 0. In that case, `updatePool` will revert, and cosequently so will all functions that call it, including:

- deposit
- withdraw
- massUpdatePools
- updateBoost

Exploit scenario

Eve is a malicious user listening to transactions for the deployed BoostedMasterChefJoe contract. As soon as she spots a call to `add` to add a new lp token, she procures that token and sends a negligible amount to `bmcj`. As a result, she causes DoS on this token without possibility to remediate it by the owner or community.

Recommendation

Short term, change the logic of the contract to take into account the possibility of malicious actors sending small amounts of tokens to it.

Long term, make use of (and build on top of) the fuzzing model. This will ensure issues like this are identified during testing.

Go back to Findings Summary

H3: Many components lack data validation

Impact:	High	Likelihood:	Low
Target:	MoneyMaker, StableJoeStaking	Type:	Data validation

Listing 4. MoneyMaker.constructor

```
/// @notice Constructor
/// @param _factory The address of JoeFactory
/// @param _bar The address of JoeBar
/// @param _tokenTo The address of the token we want to convert to
/// @param _wavax The address of wavax
constructor(
    address _factory,
    address _bar,
    address _tokenTo,
    address _wavax
) public {
    factory = IJoeFactory(_factory);
    bar = _bar;
    tokenTo = _tokenTo;
    wavax = _wavax;
    devAddr = msg.sender;
    isAuth[msg.sender] = true;
    authorized.push(msg.sender);
}
```

Listing 5. StableJoeStaking.initialize

```
    function initialize(
        IERC20Upgradeable _rewardToken,
        IERC20Upgradeable _joe,
        address _feeCollector,
        uint256 _depositFeePercent
    ) external initializer {
        __Ownable_init();
        require(_feeCollector != address(0), "StableJoeStaking: fee collector can't be address 0");
        require(_depositFeePercent <= 5e17, "StableJoeStaking: max deposit fee can't be greater than 50%");

        joe = _joe;
        depositFeePercent = _depositFeePercent;
        feeCollector = _feeCollector;

        isRewardToken[_rewardToken] = true;
        rewardTokens.push(_rewardToken);
        PRECISION = 1e24;
    }
```

Description

Many components in the system lack appropriate data validation such as zero-address checks (see Listing 4 and Listing 5). While not a perfect method of data validation, zero-address checks are the first line of defense against incorrectly supplied input arguments.

Vulnerability scenario

Bob is an employee of Trader Joe or a project building on top of Trader Joe. He initializes StableJoeStaking, but because of a bug in the scripting library, the abi values are incorrectly encoded. As a result, the `joe` storage variable is set to `address(0)`, leading to unintended consequences.

Recommendation

Short term, add a zero-address check for all addresses and contracts used as inputs to the system.

Long term, investigate more stringent method of data validation, such as through a specific id, to catch even more instances of machine or human error.

Go back to Findings Summary

H4: Renounce ownership

Impact:	High	Likelihood:	Low
Target:	VeJoeToken	Type:	Access Control

Description

For staking purposes, `owner` must be set to VeJoeStaking contract address. Therefore, `renounceOwnership` is not potentially useful.

Exploit scenario

`owner` is renounced, and thus users of VeJoeStaking can not claim their veJOE or JOE tokens.

Recommendation

Override the `renounceOwnership` method to disable this unwanted feature.

Go back to Findings Summary

H5: `setBoostedMasterChefJoe` has insufficient data validation

Impact:	High	Likelihood:	Low
Target:	VeJoeToken	Type:	Data validation

Description

`VeJoeToken` does not perform any data validation whatsoever of `_boostedMasterChef` in its `setBoostedMasterChefJoe` function.

As a consequence of using token hooks (`_afterTokenOperation`), there is a risk that incorrect value can block minting, burning, or cause malicious behavior.

Exploit scenario

An incorrect or malicious `_boostedMasterChef` is passed it. Instead of reverting, the call succeeds.

Recommendation

Add more stringent data validation for `_boostedMasterChef`. We recommend defining a getter such as `contractType()` that would return a hash of an identifier unique to the (project, contract) tuple (an example would be `keccak256("Trader Joe: Boosted Master Chef")`). This will ensure the call reverts for most incorrectly passed values. However, only if they are passed by accident. Incorrect values that are passed intentionally can succeed (viz Trust Model).

Go back to Findings Summary

H6: Tokens with callbacks

Impact:	High	Likelihood:	Low
Target:	/**/*	Type:	Token interaction, Re-entrancy

Listing 6. MoneyMaker._swap#L331

```
        IERC20(fromToken).safeTransfer(address(pair), amountIn);
```

Listing 7. MoneyMaker._swap#L351

```
        pair.swap(amount0Out, amount1Out, to, new bytes(0));
```

Listing 8. StableJoeStaking.sol#L134-L143

```
            if (_previousAmount != 0) {
                uint256 _pending = _previousAmount.mul(accRewardPerShare[_token]).div(PRECISION).sub(
                    user.rewardDebt[_token]
                );
                if (_pending != 0) {
                    safeTokenTransfer(_token, msg.sender, _pending);
                    emit ClaimReward(msg.sender, address(_token), _pending);
                }
            }
            user.rewardDebt[_token] = _newAmount.mul(accRewardPerShare[_token]).div(PRECISION);
```

Description

There are many situations in the codebase when token transfers are done in the middle of a state-changing function (see Listing 6 together with Listing 7, or Listing 8). If the tokens transferred have callbacks (e.g. all ERC223 and

ERC777 tokens), this might create re-entrancy possibilities.

Exploit scenario

A token with callbacks is entered as a parameter either to MoneyMaker, either as an input to `_convert`, or as a bridge for another token, or to StableJoeStaking as a reward token. As a result, a re-entrancy can be executed.

Recommendation

Ensure that no tokens with callbacks are added, either:

- as reward tokens in StableJoeStaking,
- as LP tokens in MasterChefJoeV2 or BoostedMasterChefJoe,
- or to be supplied as user input in MoneyMaker.

This will ensure the system is resilient against re-entrancy attacks.

Go back to Findings Summary

H7: Usage of `solc` optimizer

Impact:	High	Likelihood:	Low
Target:	`/**/*`	Type:	Compiler configuration

Description

The project uses the `solc` optimizer. Enabling the `solc` optimizer may lead to unexpected bugs.

The Solidity compiler was audited in November 2018 and the audit concluded that the optimizer may not be safe.

Vulnerability scenario

A few months after deployment, a vulnerability is discovered in the optimizer. As a result, it is possible to attack the protocol.

Recommendation

Until the `solc` optimizer undergoes more stringent security analysis, opt out using it. This will ensure the protocol is resilient to any existing bugs in the optimizer.

Go back to Findings Summary

M1: Setting `anyAuth` to `true` leads to undefined behavior

Impact:	Medium	Likelihood:	High
Target:	MoneyMaker	Type:	Access controls

Listing 9. MoneyMaker.setAnyAuth

```
    function setAnyAuth(bool access) external onlyOwner {
        anyAuth = access;
    }
```

Listing 10. MoneyMaker.onlyAuth

```
    modifier onlyAuth() {
        require(isAuth[msg.sender] || anyAuth, "MoneyMaker: FORBIDDEN");
        _;
    }
```

Listing 11. MoneyMaker.setBridge

```
    function setBridge(address token, address bridge) external onlyAuth {
        // Checks
        require(token != tokenTo && token != wavax && token != bridge, "MoneyMaker: Invalid bridge");

        // Effects
        _bridges[token] = bridge;
        emit LogBridgeSet(token, bridge);
    }
```

Description

MoneyMaker allows the owner to set `anyAuth` to `true` (see Listing 9). This means anyone can call functions with the `onlyAuth` modifier (see Listing 10).

This means that anyone can set bridges (see Listing 11).

Exploit scenario

The `owner` sets `anyAuth` to `true`. Mallory can now set a bridge to an untrusted, malicious token. Since these tokens are called in the `_swap` function, this can lead to denial of service and re-entrancy attacks.

Recommendation

Short term, set the `setBridge` function to `onlyOwner` rather than `onlyAuth`. This will ensure that the bridges store is not vulnerable to untrusted user inputs. Additionally, consider removing the `anyAuth` case altogether. Even if bridges are not vulnerable, any form of untrusted token input could lead to re-entrancy vulnerabilities.

Long term, avoid patterns where calls are made to addresses supplied by untrusted parties. This will prevent re-entrancy attacks.

Go back to Findings Summary

M3: Renewing boosting period can fail

Impact:	Medium	Likelihood:	High
Target:	VeJoeStaking	Type:	Logic fault

Description

Users can spend their JOE tokens meaninglessly if they deposit them with the thought of extending the boosting period.

Exploit Scenario

The user wants to extend his boosted period to earn more veJOE tokens. He will do it **before** the end of the current boosted period, and it will cause he will spend JOE tokens without extending it.

Recommendation

Remove the first condition in `deposit` function:

```
            if (userInfo.lastRewardTimestamp == 0) {
                userInfo.boostEndTimestamp = block.timestamp.add
(boostedDuration);
            }
```

NOTE	this issue was present in the first revision of the contracts we audited (see Executive Summary).

Go back to Findings Summary

M2: Array lengths are not validated

Impact:	Medium	Likelihood:	Medium
Target:	MoneyMaker	Type:	Data validation

Listing 12. MoneyMaker.convertMultiple

```
    /// @notice Converts a list of pairs of tokens to tokenTo
    /// @dev _convert is separate to save gas by only checking the 'onlyEOA' modifier once in case of convertMultiple
    /// @param token0 The list of addresses of the first token of the pairs that will be converted
    /// @param token1 The list of addresses of the second token of the pairs that will be converted
    /// @param slippage The accepted slippage, in basis points aka parts per 10,000 so 5000 is 50%
    function convertMultiple(
        address[] calldata token0,
        address[] calldata token1,
        uint256 slippage
    ) external onlyEOA onlyAuth {
        // TODO: This can be optimized a fair bit, but this is safer and simpler for now
        require(slippage < 5_000, "MoneyMaker: slippage needs to be lower than 50%");

        uint256 len = token0.length;
        for (uint256 i = 0; i < len; i++) {
            _convert(token0[i], token1[i], slippage);
        }
    }
```

Description

There are multiple times when publicly-entrypoints accept multiple arrays as parameters. In many cases, there is no check to ensure the lengths are equal (see Listing 12).

Vulnerability scenario

Due to a bug in a scripting library, Alice's transaction is encoded with `token1` having more values that `token0`. The `token1` values are never executed, leading to unintended consequences.

Recommendation

Short term, either add data validation to such cases to ensure that the lengths of the arrays are the same, or mark the function as low-level using natspec documentation, and create a `periphery` contract that users are expected to interact with.

Long term, ensure contracts are resilient to human and machine error.

Go back to Findings Summary

L1: Code duplication in MoneyMaker

Impact:	Low	Likelihood:	N/A
Target:	MoneyMaker	Type:	Code duplication

Listing 13. MoneyMaker.getAmountOut

```
    function getAmountOut(
        uint256 amountIn,
        uint256 reserveIn,
        uint256 reserveOut
    ) internal pure returns (uint256 amountOut) {
        require(amountIn > 0, "MoneyMaker: INSUFFICIENT_INPUT_AMOUNT");
        require(reserveIn > 0 && reserveOut > 0, "MoneyMaker: INSUFFICIENT_LIQUIDITY");
        uint256 amountInWithFee = amountIn.mul(997);
        uint256 numerator = amountInWithFee.mul(reserveOut);
        uint256 denominator = reserveIn.mul(1000).add(amountInWithFee);
        amountOut = numerator / denominator;
    }
```

Listing 14. JoeLibrary.getAmountOut

```
    function getAmountOut(
        uint256 amountIn,
        uint256 reserveIn,
        uint256 reserveOut
    ) internal pure returns (uint256 amountOut) {
        require(amountIn > 0, "JoeLibrary: INSUFFICIENT_INPUT_AMOUNT");
        require(reserveIn > 0 && reserveOut > 0, "JoeLibrary: INSUFFICIENT_LIQUIDITY");
        uint256 amountInWithFee = amountIn.mul(997);
        uint256 numerator = amountInWithFee.mul(reserveOut);
        uint256 denominator = reserveIn.mul(1000).add(amountInWithFee);
        amountOut = numerator / denominator;
    }
```

Description

The function `MoneyMaker.getAmountOut` is equivalent to the function `JoeLibrary.getAmountOut` (see Listing 13 and Listing 14). This is code duplication that increases maintenance costs and chance of bugs.

Recommendation

Short term, use `JoeLibrary.getAmountOut` in `MoneyMaker`. If only one function of a library is used, the bytecode will be inserted into the calling contract (in this case MoneyMaker), so there will be no performance trade-off.

Long term, avoid code duplication where possible. This will prevent bugs in the future.

Go back to Findings Summary

W1: OpenZeppelin dependencies contain bugs

Impact:	Warning	Likelihood:	N/A
Target:	/node_modules/@openzeppelin/{contracts,contracts-upgradeable}	Type:	Dependencies

Listing 15. package.json's OpenZeppelin dependencies

```
"@openzeppelin/contracts": "^3.1.0",
"@openzeppelin/contracts-upgradeable": "3.3.0",
```

Description

Currently, the project uses `@openzeppelin/contracts` at `^3.1.0` and `@openzeppelin/contracts-upgradeable` at `3.3.0` (see Listing 15). These versions are known to have numerous vulnerability, including:

- Initializer reentrancy may lead to double initialization
- TimelockController vulnerability in OpenZeppelin Contracts

We did not find instances of these vulnerabilities in the codebase, nevertheless, we would recommend to use the latest dependency versions.

Recommendation

Short term, update the dependencies' versions to the latest version (`^4.5.0` as of the this writing). This will ensure fewest possible bugs in the dependencies are present.

Long term, update dependency versions often to ensure the latest version is used. Additionally, pay special attention to security advisory banks of dependencies.

Go back to Findings Summary

W2: Front-runnning initialize function

Impact:	Warning	Likelihood:	N/A
Target:	VeJoeStaking	Type:	Front-running

Description

An attacker can front-run the initialization of a newly created contract and call arbitrary functions in it.

Exploit scenario

Alice just deployed the contract and wants to call `initialize` function. Bob noticed the deploy and front-runs Alice's initialization transaction, which will give him control over the contract.

Recommendation

Add `initializer` modifier on the constructor in VeJoeStaking or do atomic upgrades (contract creation and calling `initialize` in one transaction).

Go back to Findings Summary

W3: `getPendingVeJoe` does not have up to date values

Impact:	Warning	Likelihood:	N/A
Target:	VeJoeStaking	Type:	Function behaviour

Description

`getPendingVeJoe` does not contain a call to `updateRewardVars` and thus its results can be outdated.

Recommendation

If `getPendingVeJoe` needs to be publicly accessible, then add `updateRewardVars` to the function. Otherwise, set it private.

Go back to Findings Summary

W4: Pre-0.8 `solc` versions don't check for arithmetic overflow

Impact:	Warning	Likelihood:	N/A
Target:	`/**/*`	Type:	Compiler configuration

Description

Versions of the `solc` compiler prior to `0.8.0` do not check for arithmetic overflows and underflows of integer types.

Recommendation

We recommend using `0.8.0` at minimum, but the newest one is also not recommended. A good practice is the latest compiler version roughly 3-6 months old.

Go back to Findings Summary

I1: `MoneyMaker.authorized` keeps old values

Impact:	Informational	Likelihood:	High
Target:	MoneyMaker	Type:	Data consistency

Listing 16. MoneyMaker.sol#L79-L90

```
/// @notice Adds a user to the authorized addresses
/// @param _auth The address to add
function addAuth(address _auth) external onlyOwner {
    isAuth[_auth] = true;
    authorized.push(_auth);
}

/// @notice Remove a user of authorized addresses
/// @param _auth The address to remove
function revokeAuth(address _auth) external onlyOwner {
    isAuth[_auth] = false;
}
```

Description

MoneyMaker contains state variables `isAuth` and `authorized` (a `mapping` and `array` repectively), which track authorization of addresses to call protected functions. When a new address is added, both variables are updated. However, when one is removed, `authorized` never gets updated. This is compounded by the fact that `authorized` is a `public` variable.

Vulnerability scenario #1

A protocol built on top of Trader Joe reads `authorized`, expecting that it holds the current values. This can lead to unintended consequences.

Vulnerability scenario #2

A Trader Joe developer is building a new version of this module. He makes an authorization check that involves reading from `authorized`. Old values are kept, leading to data inconsistency.

Recommendation

Short term, investigate alternative data structures that would allow efficient storage of authorized addresses. Examples include implementing linked lists using mappings or using OpenZeppelin's EnumerableSet.

Long term, avoid pattern with inconsistent data. This will prevent bugs further down the line.

Go back to Findings Summary

I2: Use `_msgSender` over `msg.sender`

Impact:	Informational	Likelihood:	N/A
Target:	`/**/*`	Type:	Builtin variables

Description

Many contracts, e.g. MoneyMaker, have Context or ContextUpgradeable in their inheritance chain. Context and ContextUpgradeable define the `_msgSender` and `_msgData` functions. This makes it easy to switch their semantics, e.g. if Trader Joe decides to support metatransactions in the future. If a contract inherits from `Context` or (or `ContextUpgradeable`), uses of `msg.data` and `msg.sender` should be replaced by `internal` calls to `_msgData` and `_msgSender`, respectively. This will ensure that if the semantics is changed in the future, the codebase will remain consistent.

Recommendation

Short term, replace all instances of `msg.sender` with `_msgSender()` in the contracts that inherit from Context or ContextUpgradeable. This will ensure future-proofness against future code changes.

Long term, ensure that all contracts' code is consistent with the code of their inherited contracts.

I3: Log old values in logs

Impact:	Informational	Likelihood:	High
Target:	/**/*	Type:	Logging

Listing 17. MoneyMaker.setBridge

```
    /// @notice Force using `pair/bridge` pair to convert `token`
    /// @param token The address of the tokenFrom
    /// @param bridge The address of the tokenTo
    function setBridge(address token, address bridge) external onlyAuth
{
        // Checks
        require(token != tokenTo && token != wavax && token != bridge,
"MoneyMaker: Invalid bridge");

        // Effects
        _bridges[token] = bridge;
        emit LogBridgeSet(token, bridge);
    }
```

Description

When logging important state changes, currently the codebase usually logs only the new value (see Listing 17). This might make incident analysis and other analyses of runtime behavior difficult.

Recommendation

Short term, log old values for very important operations such as updating implementation pointers. This will ensure the most possible information is available for someone analyzing runtime behavior.

Long term, log any values that on-chain and off-chain observers might be interested in. This ensures the maximum transparency of the protocol to its users, developers and other stakeholders.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Trader Joe, March 3, 2022.

If an individual issue is referenced, please use the following identifier:

`ABCH-{project_identifer}-{finding_number}`,

where `{project_identifier}` for this project is `TRADER-JOE` and `{finding-number}` is the integer corresponding to the section number aligned to three digits. For example, to cite I1: `MoneyMaker.authorized` keeps old values, we would use `ABCH-TRADER-JOE-001`.

Appendix B: Glossary of terms

The following terms might be used throughout the document:

Public entrypoint

An `external` or `public` function.

Publicly-accessible function/entrypoint

An `external` or `public` function that can be successfully executed by any network account.

Appendix C: Non-Security-Related Recommendations

C.1. Variables can be made immutable/constant

There are several variables in the contracts that are assigned once in the initialization function without an option to be changed. These include:

- StableJoeStaking:
  a. `joe`
  b. `feeCollector`
  c. `PRECISION`
- VeJoeStaking:
  a. `ACC_VEJOE_PER_SHARE_PRECISION`
  b. `VEJOE_PER_SHARE_PER_SEC_PRECISION`
- BoostedMasterChefJoe:
  a. `MASTER_CHEF_V2`
  b. `JOE`
  c. `VEJOE`
  d. `MASTER_PID`
  e. `ACC_TOKEN_PRECISION`
  f. `maxBoostFactor`

If the variables were declared `constant` or `immutable`, the values would be stored as constant expressions in the logic contract's code. Because they would be part of the contract's code, their values would be visible even in calls from a proxy contract. To retain the ability to parameterize the value of the variables, the variables should be declared `immutable` (constants are replaced at compile time).

This change would save much gas because the variables would not have to be read from the storage.

Appendix D: Upgradeability

There are three topics pertaining to security currently in upgradeability:

1. Access controls on logic contracts to prevent malicious actors from interacting with them directly. Note that this is only a problem insofar as they could change the logic contract's code.
2. An attacker calling other functions on the Proxy before initialize is called on it.
3. An attacker front-running one of the initialization functions.

Contract code invariant	A contract that doesn't use `callcode`, `delegatecall` or `selfdestruct` instructions cannot be self-destructed. Moreover, its code cannot change.

Based on the Contract code invariant, the only way to change a contract's code is through the use of `callcode`, `delegatecall` or `selfdestruct`.

The best way to accomplish both (1) and (2) (while preserving (3)) is to:

1. Ensure that no function on the logic contract can be called until its initialization function is called.
2. Make sure that once the logic contract is constructed, its initialization function cannot be called.
3. Ensure that the initialization function can be called on the Proxy.
4. Ensure that all functions can be called on the Proxy once it has been initialized.

If we are able to accomplish these (and only these) constraints, then the only risk will be the front-running of the initialization function by an attacker; we'll inspect that later.

The initialization function can only currently be called once. Hence the way to accomplish the above (and only the above) constraints is to:

1. Add the `initialized` modifier to the constructor of the logic contract. The constructor will be called on the logic, but not on the proxy contract (see Listing 18)
2. Add a `initializer` storage slot that gets set to `true` on initialization (see Listing 19). Note that we have to define a new variable since OpenZeppelin's `_initialized` is marked as `private`. Add a require to every non-view public entry point in the logic contract that it has been initialized (see Listing 20).

Listing 18. To be added to the logic contract

```
bool public initialized;

constructor() initializer {}
```

Listing 19. To be added to `initialize` on the logic contract

```
initialized = true;
```

Listing 20. To be added to every non-view public entrypoint on the logic contract

```
modifier onlyInitialized() {
    require(initialized);
    _;
}
```

In summary, the process would be to:

1. Add a requirement to every non-view public entrypoint that the contract has been initialized.

2. Add a requirement to the initialization function that it cannot be called on the logic contract.

Together, these will accomplish both (1) and (2) of the upgradeability requirements.

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://discord.gg/z4KDUbuPxq