Trader Joe

New BoostedMasterChefJoe

18 March 2022

by Ackee Blockchain

Contents

1. Document Revisions

1.0	Final report	March 18, 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 Šmolík	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 March 14 and March 18, 2022, Trader Joe engaged ABCH to conduct a security review of the new BoostedMasterChefJoe contract. This was a follow-up assessment from our earlier audit, where we reviewed the old BoostedMasterChefJoe, among others.

Working from commit 27c7c77c39, we were allocated 5 engineering days and the lead auditor was Dominik Teiml.

We began our review by looking for common Solidity pitfalls. This yielded several issues such as M2: `set` performs `sub` before `add` and W2: Tokens with callbacks. 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 contract ensured?
- Do the contracts correctly use dependencies or other contracts they rely on, such as OpenZeppelin dependencies?
- 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?

Our review resulted in 7 findings, ranging from Informational to High severity. The most severe one could cause contract insolvency if the `set` function is front-run (see H1: Front-running `set` could lead to insolvency).

Ackee Blockchain recommends Trader Joe:

- expand on our earlier fuzzing model to heavily test the new BoostedMasterChefJoe contract,
- 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

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.

4.2. Actors

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 share of Joe reserved for boosted rewards. They can also transfer and renounce ownership.

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: Front-running `set` could lead to insolvency	Arithmetic, Front-running	High	Medium
M1: `pendingTokens` may unexpectedly revert	Arithmetic	Medium	Medium
M2: `set` performs `sub` before `add`	Arithmetic	Low	Medium
M3: Usage of `solc` optimizer	Compiler configuration	High	Low
W1: OpenZeppelin dependencies contain bugs	Dependencies	Warning	N/A

	Type	Impact	Likelihood
W2: Tokens with callbacks	Token interaction, Re-entrancy	Warning	N/A
I1: Use `_msgSender` over `msg.sender`	Builtin variables	Informational	N/A

Table 1. Table of Findings

H1: Front-running `set` could lead to insolvency

Impact:	High	Likelihood:	Medium
Target:	BoostedMasterChefJoe	Type:	Arithmetic, Front-running

Listing 1. Excerpt from BoostedMasterChefJoe.set

```
    function set(
        uint256 _pid,
        uint96 _allocPoint,
        uint32 _veJoeShareBp,
        IRewarder _rewarder,
        bool _overwrite
    ) external onlyOwner {
        require(_veJoeShareBp <= 10_000, "BoostedMasterChefJoe: veJoeShareBp needs to be lower than 10000");
        PoolInfo storage pool = poolInfo[_pid];
        totalAllocPoint = totalAllocPoint.sub(pool.allocPoint).add(_allocPoint);
        pool.allocPoint = _allocPoint;
        pool.veJoeShareBp = _veJoeShareBp;
        if (_overwrite) {
            if (address(_rewarder) != address(0)) {
                // Sanity check
                _rewarder.onJoeReward(address(0), 0);
            }
            pool.rewarder = _rewarder;
        }

        massUpdatePools();
```

Description

`set` can be used by the owner to change the `allocPoint`, `veJoeShareBp` and `rewarder` parameters of a pool. `massUpdatePool` is a function that calls `updatePool` on all pools, which updates `accJoePerShare` and `accJoePerFactorPerShare`.

When `set` is called, it first applies changes to `allocPoint` and only then calls `massUpdatePool` (see Listing 1). Since each pool keeps track of its own `lastRewardTimestamp`, it is possible for an attacker to front-run this call with a call to `updatePool` of the pool whose share will get decreased with the call to `set`.

As a result, this pool's rewards will be calculated using the old share, while other pools' rewards will then be calculated in `massUpdatePool` using the new ratios. Since these ratios might differ, it can lead to an unexpectedly high value being assigned to `accJoePerShare` or `accJoePerFactorPerShare`. As a result, the contract might be unavailable for withdrawals or deposits, as it is unable to cover its debts.

Exploit scenario

See Description.

Recommendation

Short term, move the call to `massUpdatePools` before any other state assignments are performed. This will ensure all pools have their rewards calculated according to the same share.

Long term, ensure the contracts are resilient against front-running. This will ensure issues like this don't come up in the future.

Go back to Findings Summary

M1: `pendingTokens` may unexpectedly revert

Impact:	Medium	Likelihood:	Medium
Target:	BoostedMasterChefJoe	Type:	Arithmetic

Listing 2. Excerpt from BoostedMasterChefJoe.pendingTokens

```
            accJoePerFactorPerShare = accJoePerFactorPerShare.add(
                joeReward.mul(ACC_TOKEN_PRECISION).mul(pool
.veJoeShareBp).div(pool.totalFactor.mul(10_000))
            );
```

Description

`pendingTokens` is a `view` function that calculates a user's pending rewards. In one expression, it performs division by `pool.totalFactor` (see Listing 2). If no user of the pool owns veJoe tokens, the total factor will be 0. Even though a user may have pending rewards, the function will revert.

Exploit scenario

Alice is a user of the protocol. She queries the contract to retrieve her pending tokens, but instead, the call reverts. This can lead to unintended consequences.

Recommendation

Short term, wrap the assignment expression expression in an if clause, to ensure it is executed only given correct preconditions.

Long term, ensure all arithmetic expressions give correct values under all possible states of the contracts.

Go back to Findings Summary

M2: `set` performs `sub` before `add`

Impact:	Low	Likelihood:	Medium
Target:	BoostedMasterChefJoe	Type:	Arithmetic

Listing 3. Excerpt from BoostedMasterChefJoe.set

```
    function set(
        uint256 _pid,
        uint96 _allocPoint,
        uint32 _veJoeShareBp,
        IRewarder _rewarder,
        bool _overwrite
    ) external onlyOwner {
        require(_veJoeShareBp <= 10_000, "BoostedMasterChefJoe: veJoeShareBp needs to be lower than 10000");
        PoolInfo storage pool = poolInfo[_pid];
        totalAllocPoint = totalAllocPoint.sub(pool.allocPoint).add(_allocPoint);
```

Description

BoostedMasterChefJoe contains a function `set` which can update the `allocPoint`, `veJoeShareBp` and `rewarder` of a pool. When updating the global `totalAllocPoint`, it subtracts before it adds.

Exploit scenario

There are two initialized pools, both with allocation points 100. The owner wants to change one pool to be 3x more significant than the other. He calls `set` with 300. Instead of currectly updating the allocation points, the function call reverts.

Recommendation

Short term, add before subtracting in the function above.

Long term, ensure that the contracts have expected behavior for all ranges of valid inputs.

Go back to Findings Summary

M3: 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

W1: OpenZeppelin dependencies contain bugs

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

Listing 4. Excerpt from 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 4). 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: Tokens with callbacks

Impact:	Warning	Likelihood:	N/A
Target:	BoostedMasterChefJoe.sol	Type:	Token interaction, Re-entrancy

Listing 5. Excerpt from BoostedMasterChefJoe.emergencyWithdraw

```
        user.factor = 0;

        IRewarder _rewarder = pool.rewarder;
        if (address(_rewarder) != address(0)) {
            _rewarder.onJoeReward(msg.sender, 0);
        }

        // Note: transfer can fail or succeed if `amount` is zero
        pool.lpToken.safeTransfer(msg.sender, amount);
```

Listing 6. Excerpt from SimpleRewarderPerSec.onJoeReward

```
                if (pending > balance) {
                    rewardToken.safeTransfer(_user, balance);
                    user.unpaidRewards = pending - balance;
                } else {
```

Description

There are situations in the codebase when token transfers are done in the middle of a state-changing function (see Listing 5 and Listing 6). 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 reward token to SimpleRewarderPerSec.

As a result, a re-entrancy can be executed.

Recommendation

Ensure that no tokens with callbacks are added to the system. This will ensure the system is resilient against re-entrancy attacks.

Go back to Findings Summary

I1: Use `_msgSender` over `msg.sender`

Impact:	Informational	Likelihood:	N/A
Target:	BoostedMasterChefJoe	Type:	Builtin variables

Description

BoostedMasterChefJoe has ContextUpgradeable in its inheritance chain. ContextUpgradeable defines 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 `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.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, Trader Joe 2, March 18, 2022.

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

`ABCH-{project_identifer}-{finding_id}`,

where `{project_identifier}` for this project is `TRADER-JOE-02` and `{finding-id}` is the (severity, count) combination that appears as the prefix of the issue. For example, to cite an issue with a prefix `M3`, we would use `ABCH-TRADER-JOE-02-M3`.

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. Bits vs bytes

Listing 7. Excerpt from [BoostedMasterChefJoe.sol#L50-L50](#)

```
50        // Address are stored in 160 bytes, so we store allocPoint in 96 bytes to
```

[BoostedMasterChefJoe](#)'s code comments mention that addresses are stored in 160 bytes of data, and seem to assume that Solidity types `uintX` use `X` bytes (see [Listing 7](#)). In fact, addresses are stored in 160 bits, and the above set of Solidity types uses `X` bits.

C.2. `set` contains an unnecessary assignment

Listing 8. Excerpt from [BoostedMasterChefJoe.set](#)

```
        PoolInfo storage pool = poolInfo[_pid];
        totalAllocPoint = totalAllocPoint.sub(pool.allocPoint).add(_allocPoint);
        pool.allocPoint = _allocPoint;
        pool.veJoeShareBp = _veJoeShareBp;
        if (_overwrite) {
            if (address(_rewarder) != address(0)) {
                // Sanity check
                _rewarder.onJoeReward(address(0), 0);
            }
            pool.rewarder = _rewarder;
        }

        massUpdatePools();

        poolInfo[_pid] = pool;
```

C.3. Variables can be made immutable/constant

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

1. `MASTER_CHEF_V2`
2. `JOE`
3. `VEJOE`
4. `MASTER_PID`
5. `ACC_TOKEN_PRECISION`

if the contract had a constructor and the variables were declared `constant` or `immutable` and assigned to in the constructor, 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 9)
2. Add a `initializer` storage slot that gets set to `true` on initialization (see Listing 10). 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 11).

Listing 9. To be added to the logic contract

```
bool public initialized;

constructor() initializer {}
```

Listing 10. To be added to `initialize` *on the logic contract*

```
initialized = true;
```

Listing 11. 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