PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Trader Joe (Reward Distributor V2)

05 May 2022

paladinsec.co

info@paladinsec.co

Table of Contents

Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocation for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

Paladin retains full rights over all intellectual property (including expertise and new attack or exploit vectors) discovered during the audit process. Paladin is therefore allowed and expected to re-use this knowledge in subsequent audits and to inform existing projects that may have similar vulnerabilities. Paladin may, at its discretion, claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for Trader Joe's Reward Distributor V2 contracts on the Avalanche network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

1.1 Summary

Project Name	Trader Joe
URL	https://traderjoexyz.com/
Network	Avalanche
Language	Solidity

1.2 Contracts Assessed

Name	Contract	Live Code Match
RewardDistributorV2	0x5d52300fa52845874f06430c3b0db386aab877f9	✓ MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
● High	3	3	-	-
● Medium	0	-	-	-
● Low	1	1	-	-
● Informational	5	4	-	1
Total	**9**	**8**	**-**	**1**

Classification of Issues

Severity	Description
● High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
● Medium	Bugs or issues that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
● Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
● Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 RewardDistributorV2

ID	Severity	Summary	Status
01	HIGH	Reward inflation exploit: A strategic user can drain the new distributor's reward tokens through a strategically organized flashloan exploit	✓ RESOLVED
02	HIGH	Reentrancy risk: `_claimReward` does not adhere to checks-effects-interactions, potentially allowing the whole `RewardDistributor` to be drained if reentrancy is permitted on the tokens or AVAX transfers from the old distributor	✓ RESOLVED
03	HIGH	Double rewards issued in the new Distributor	✓ RESOLVED
04	LOW	`initialize` method can be called twice	✓ RESOLVED
05	INFO	Gas optimisations	ACKNOWLEDGED
06	INFO	Critical variables can be changed which can lead to the misconfiguration and drainage of the rewarder	✓ RESOLVED
07	INFO	Typographical errors	✓ RESOLVED
08	INFO	Lack of validation	✓ RESOLVED
09	INFO	The protocol will stop working on Sun, Feb 07, 2106	✓ RESOLVED

2 Findings

2.1 RewardDistributorV2

`RewardDistributorV2` represents the new reward distribution for the TraderJoe Lending (Banker Joe) protocol that controls the accounting and distribution within the Banker Joe system. It migrates from the old `RewardDistributor`.

Banker Joe is a lending market that allows users to supply and borrow various assets. Over time, borrowers pay interest to suppliers. Trader Joe, however, distributes additional rewards to incentivise users further. Users receive these rewards by supplying and borrowing assets.

`RewardDistributor` is responsible for the accounting and distribution of these rewards. The distributor uses logic introduced by Synthetix's `StakingRewards`. Each token market emits both AVAX and JOE amounts. Each emission token has a supply and borrowing emission rate for each token market. These emission rates represent the total number of tokens emitted every second to that side of the token lending market.

To ensure that no abuse or bad accounting is possible at a lower level, the `Joetroller` calls the `RewardDistrbutor` on every token transfer, deposit, borrow, liquidation, repay, and withdrawal. This link between the `Joetroller` and the `RewardDistrbutor` is the central driver of the rewarding mechanism.

2.1.1 Privileged Functions

The following functions can be called by the owner of the contract:

- `updateAndDistributeSupplierRewardsForToken`
- `updateAndDistributeBorrowerRewardsForToken`
- `initializeRewardAccrued`
- `lockInitializeRewardAccrued`
- `grantReward`
- `setJoe`
- `setJoetroller`
- `setAdmin`
- `setRewardSpeed`

2.1.2 Issues & Recommendations

Issue #01	Reward inflation exploit: A strategic user can drain the new distributor's reward tokens through a strategically organized flashloan exploit
Severity	● HIGH SEVERITY

Description

The second iteration of the reward distributor has a seemingly very useful mechanism where a user their first harvest will also harvest any pending rewards from the previous distributor (`RewardDistributor.sol`). This is especially useful for users who have not harvested for some time and would have lost their pending rewards after the upgrade.

This mechanism is implemented in a simple manner within the `_claimReward` function, called when the user does a harvest:

Line 473-478

```
if (!claimedFromOldRewarder[rewardType][holder]) {
    oldRewarder.claimReward(rewardType, holder);
    rewards =
rewards.add(oldRewarder.rewardAccrued(rewardType, holder));
    claimedFromOldRewarder[rewardType][holder] = true;
}
```

In non-technical terms, whenever the user harvests on the new rewarder for the first time, a `claimReward` call will be made to the old rewarder. This essentially does a harvest on the old rewarder and will store any reward tokens which were no longer available in the old contract in the `rewardAccrued` map.

These tokens are then distributed to the user. Even though the user could harvest them again in the old rewarder, the client has indicated that they will never enable or send rewards to this contract again.

However, one crucial mistake was made within this logic: the `oldRewarder` is only harvested when the user explicitly requests it.

Traditionally a rewarder contract must be harvested on any user balance increment to avoid reward inflation. If a harvest is not made, the new balance would be multiplied by the reward rate per token and the user would receive rewards on their now greater balance in hindsight.

To summarize and to understand this issue: A rewarder should always make sure that a harvest occurs before a balance change is made.

As the reward distributors use the users' their balances as stored in the joe tokens, this is especially important:

RewardDistributor.sol::263

```
uint256 supplierTokens = JToken(jToken).balanceOf(supplier);
```

RewardDistributor.sol:292

```
uint256 borrowerAmount =
div_(JToken(jToken).borrowBalanceStored(borrower),
marketBorrowIndex);
```

As readers might have guessed by now, there are other ways to adjust this balance other than the "harvest": supplying, borrowing, transferring tokens, repaying debt and withdrawing. The Banker Joe mechanism will always do a harvest on the linked `RewardDistributor` in these scenarios to ensure the issue explained above does not present itself.

What it does not do is call the old rewarder. This means that the old rewarder will have potentially increased user balances without a harvest and that when the user eventually calls the logic to harvest from the old rewarder, their new balance will be used instead of their correct old one. If the user still has pending rewards from before rewards were disabled, these pending rewards will be multiplied by the users present balance instead of their historical balance and the protocol therefore overcompensates users.

This overcompensation can be crucially exploited to drain the complete `RewardDistributorV2` of reward tokens through a flashloan exploit which is described below.

Proof of concept

1. Alice stakes 1 AVAX for 1 month with 10 different wallets before the old rewarder is disabled and accrues 0.1 pending JOE tokens on each wallet which she does not harvest (reward per staked token is 0.1).

2. Trader Joe disables the old rewarder (emission rate becomes zero) and upgrades to the new rewarder.

3. Alice flashloans 1,000,000 AVAX.

4. Allice immediately supplies it to Banker Joe. A harvest occurs on the new rewarder but does not occur on the old rewarder.

5. Alice does a manual harvest on the new rewarder which triggers a harvest on the old rewarder.

6. The old rewarder still has the stored 0.1 pending JOE but now multiplies it with 1,000,000 AVAX supplierTokens (line 263-264). Alice receives 100,000 JOE.

7. Alice unstakes the 1,000,000 AVAX and moves it to one of her other wallets to repeat steps 4, 5, 6 and 7 for all her wallets.

8. Alice repays her loan.

The total profit in this example is 1,000,000 JOE with a flash loan of 1,000,000 AVAX and a capital cost of 10 AVAX. The multiple wallet step (step 7) is not strictly necessary but is provided as a demonstration of how the flash loan cost can be arbitrarily reduced by increasing the number of wallets. The capital cost can furthermore be arbitrarily reduced as it does not matter whether Alice stakes 1 AVAX or 0.01 AVAX. It should be clear that this exploit can be organized at an arbitrarily low economical cost and should therefore be considered severe as it allows anyone with a pending balance in the old rewarder to potentially drain the new rewarder.

Recommendation

Consider consistently updating the old rewarder on any new rewarder update. Alternatively and more desirable, consider not linking the two rewarders at all and distributing old rewards in a less tied manner.

Resolution

✓ RESOLVED

The client has removed the old rewarder hook completely and will instead opt for adding the unclaimed amounts of users to the new rewarder with the new `initializeRewardAccrued` function.

The flow of upgrading to the new rewarder will be and must be as follows:

- Set oldRewarder markets speed to 0
- Drain all rewards from the oldRewarder
- Calculate all the users' unclaimed rewards off-chain
- Deploy the RewarderV2 and use the initializeRewardAccrued function to add the calculated rewards to users their rewards
- Lock the initializeRewardAccrued function using the new lockInitializeRewardAccrued
- Pause everything (borrow, mint, flashloan)
- Set RewarderV2 as the new rewarder on Joetroller
- Set markets speed on RewarderV2
- Unpause everything

Issue #02 — **Reentrancy risk: `_claimReward` does not adhere to checks-effects-interactions, potentially allowing the whole `RewardDistributor` to be drained if reentrancy is permitted on the tokens or AVAX transfers from the old distributor**

Severity

● HIGH SEVERITY

Description

When a user first claims on the new `RewardDistributor`, they also receive their remaining rewards from the old distributor. These rewards can be either JOE or AVAX. As AVAX transfers allow for reentrancy, this introduces a reentrancy vector.

`_claimReward` used to be carefully written in checks-effects-interactions as a reentrancy vector would allow a malicious party to inflate their rewards by reentering. This is because at the start of the `_claimReward` function, the pending rewards of the malicious party are cached while only at the end they are set to zero.

If the exploiter has any pending rewards, they would be cached throughout all reentrancy cycles and the exploiter can drain them multiple times.

It should be noted that this issue will likely not present itself as the old rewarder seems to use `.transfer` for AVAX transfers, which is known to not allow for reentrancy under the current gas requirements of Avalanche. If gas requirements of AVAX are ever lowered, the issue could become viable.

Recommendation

Consider adhering to checks-effects-interactions. Alternatively and more desirable, consider not linking the two rewarders at all and distributing old rewards in a less tied manner.

Resolution

✓ RESOLVED

The client has removed the old rewarder hook completely and will instead opt for adding users’ unclaimed amounts to the new rewarder using the new `initializeRewardAccrued` function.

Issue #03	Double rewards issued in the new Distributor
Severity	● HIGH SEVERITY
Description	When a user first claims on the new `RewardDistributor`, it also receives rewards left from the old distributor. This can be abused by a user who creates several wallets and deposits small amounts in them for a very long period of time after V2 is deposited. Example of attack vector Alice creates 10,000 wallets and deposits 10 JOE or AVAX in each one of them through the old RewardDistributor. After a year, she claims for all 10,000 wallets the rewards through the `RewardDistributorV2`, allowing her to receive double rewards — the ones from the old distributor and the new distributor.
Recommendation	Consider introducing a time-based claiming for old rewards. It could be an implemention which requires everyone to claim their rewards in the first 2 months or they will not be able claim after that period has passed.
Resolution	✓ RESOLVED The client has removed the old rewarder hook completely and will instead opt for adding users' unclaimed amounts to the new rewarder using the new `initializeRewardAccrued` function.

Issue #04	`initialize` method can be called twice
Severity	● LOW SEVERITY
Description	The `initialize` method that sets the `admin`, `joetroller` and the `oldRewarder` can be called twice which breaks the pattern of `initialize` being called just once. Additionally, this can lead to issues as `initialize` sets the `admin` and the `joetroller` by using `msg.sender`.
Recommendation	Move to an `initialize` method that can be called only once and accepts `admin` and `joetroller` as parameters.
Resolution	✓ RESOLVED The client has indicated that they will be extremely careful with their deployment setup and validate the whole setup flow. The client has explained to Paladin how this double initialization is desirable due to the overlap between a proxy initialize and joetroller initialize function.

Issue #05	Gas optimisations
Severity	● INFORMATIONAL
Description	The codebase can be slightly optimized with regards to gas usage: Usage of `uint8` and 'smaller' types is discouraged outside of structs as such small types outside of structs in fact increase the gas usage.
Recommendation	Consider replacing the `rewardType` types to `uint256`.
Resolution	● ACKNOWLEDGED

Issue #06 — Critical variables can be changed which can lead to the misconfiguration and drainage of the rewarder

Severity

INFORMATIONAL

Description

The rewarder allows the admin to change crucial variables. Most prominently, the `joetroller` address can be changed via `setJoetroller`. Changing the `joetroller` could have devastating consequences as a new `joetroller` in our opinion should almost certainly entail a new rewarder deployment as well.

There is also the secondary configuration risk of accidentally setting the emission rate to an extremely high value by Trader Joe.

Recommendation

Consider being extremely careful and consulting an auditor before ever calling `setJoetroller`. Consider capping the emission rates within the `setRewardSpeed` function to reasonable levels or consider having good governance processes in place where any function call is carefully validated.

Finally, consider limiting the number of emission tokens in the rewarder to a low level of tokens. In the past, rewarders (most notably Compound's) have been exploited through an upgrade to drain rewards that were in them. If these rewards were sufficiently low, there would not have been as much damage as there was.

Consider limiting the rewards in the rewarder (especially the first weeks after upgrading) to one or two weeks of emissions. A drip mechanism can be introduced through a secondary smart contract.

Resolution

✓ RESOLVED

The client has indicated that a MultiSig will be owner of the contracts and that all calls to the contract and especially to these functions will be inspected carefully be each signer.

Issue #07	Typographical errors
Severity	INFORMATIONAL

Description

The contract contains several typographic mistakes that we've enumerated below in a single issue to keep the report size reasonable.

Line 8

```
import "./JToken.sol";
```

The codebase presently imports the whole JToken instead of just the interface. Referencing the complete contract unnecessarily increases the verified code size in the explorer.

Line 121

```
require(msg.sender == address(joetroller) || msg.sender ==
admin, "only joe troller or admin");
```

This comment should indicate joetroller as we would not want users to think "joe" is being "trolled".

Line 228

```
* @notice Refactored function to calc and rewards accounts
supplier rewards
```

This comment still indicates supplier rewards even though it is the function related to the borrower rewards.

Lines 265-269

```
* @notice Transfer JOE to the recipient
* @dev Note: If there is not enough JOE, we do not perform
the transfer at all.
* @param recipient The address of the recipient to transfer
JOE to
* @param amount The amount of JOE to (possibly) transfer
```

This function can also transfer AVAX to the recipient.

Lines 317-321 (example)

```
function _setRewardSupplySpeed(
        uint8 rewardType,
        address jToken,
        uint256 newRewardSupplySpeed
    ) private {
```

Throughout the contract, some methods accept a `jToken` address as a parameter and some that accept it as an interface. The client should consider consistently accepting them as interfaces to clean up the codebase.

Line 506

```
* @param jTokens The market to return the pending JOE/AVAX
reward in
```

This is an array so this comment should say “markets”.

Line 642

```
joe.transfer(user, amount);
```

The result `boolean` of this transfer is not processed. As JOE always returns `true` this is not an issue in this contract but could cause problems if the contract is ever forked.

Recommendation	Consider fixing the typographical errors.
Resolution	✓ RESOLVED

Issue #08 Lack of validation

Severity ● INFORMATIONAL

Description

The contract contains functions with parameters which are not properly validated. Having unvalidated parameters could allow the governance or users to provide variable values which are unexpected and incorrect. This could cause side-effects or worse exploits in other parts of the codebase.

Consider validating the following function parameters:

Lines 271-275

```
function grantReward(
    uint8 rewardType,
    address payable recipient,
    uint256 amount
) external onlyAdmin {
```

`rewardType` lacks the validating modifier within this function. This is not a big deal but can be fixed in line with consistency.

Recommendation Consider validating the function parameters mentioned above.

Resolution ✓ RESOLVED

Issue #09	The protocol will stop working on Sun, Feb 07, 2106
Severity	● INFORMATIONAL
Location	Line 380 `rewardSupplyState[rewardType][jToken].timestamp = _safe32(_getBlockTimestamp());`
Description	As the protocol casts timestamps to 32 bits, these casts will at some point revert once they hit the 32 bit limit. This happens in the year 2106. All user stakes will become stuck at this point in time.
Recommendation	Consider this issue if this rewarder survives until the year 2106.
Resolution	✓ RESOLVED The client has increased the size of the `timestamp` variable causing this issue to no longer be present.

PALADIN

BLOCKCHAIN SECURITY