PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Covenant

23 June 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 Covenant on the Polygon 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	Covenant
URL	http://covenant.vote/
Platform	Polygon
Language	Solidity

1.2 Contracts Assessed

Name	Contract	Live Code Match
`OptimisticDistributor`	`0xE53B310431157eea3592f8b4DE92dF0Bed429139`	MATCH
`MerkleDistributor`	`0x2E2b1ba85F1110A793fC97b6AbA9dcb7Fc8bc346`	MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	1	1	-	-
Medium	2	2	-	-
Low	4	3	-	1
Informational	11	9	-	2
Total	**18**	**15**	**-**	**3**

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 Global Issues

ID	Severity	Summary	Status
01	INFO	`MerkleDistributor` and `OptimisticDistributor` do not support fee-on-transfer tokens	ACKNOWLEDGED

1.3.2 OptimisticDistributor

ID	Severity	Summary	Status
02	MEDIUM	Users can call `createReward` with any whitelisted price identifier	✓ RESOLVED
03	MEDIUM	Gas grief risk: The `priceDisputed` callback might not be guaranteed to execute which could permanently lock-in funds	✓ RESOLVED
04	LOW	`ipfsHash` and `customAncillaryData` are private	✓ RESOLVED
05	LOW	There might be a race condition between `increaseReward` and `proposeDistribution`	ACKNOWLEDGED
06	INFO	Unused import: `StoreInterface.sol`	✓ RESOLVED
07	INFO	`bondToken` can be made immutable	✓ RESOLVED
08	INFO	Lack of validation	✓ RESOLVED
09	INFO	Typographical errors	✓ RESOLVED
10	INFO	An unprivileged `syncUmaEcosystemParams` function might be undesirable	✓ RESOLVED
11	INFO	Checks-effects-interactions is not adhered to within various functions	✓ RESOLVED

1.3.3 MerkleDistributor

ID	Severity	Summary	Status
12	HIGH	The MerkleDistributor unnecessarily allows for separate rewards to be stolen from each other if a single merkle tree allows to withdraw more rewards than were provided to that tree which allows for multiple exploit vectors within the Covenant system	✓ RESOLVED
13	LOW	Governance risk: Governance can pre-configure windows with malicious Merkle trees before transferring ownership	✓ RESOLVED
14	LOW	`ipfsHash` is private	✓ RESOLVED
15	INFO	deleteWindow lacks validation	ACKNOWLEDGED
16	INFO	Gas optimizations	✓ RESOLVED
17	INFO	Unnecessary import: SafeMath.sol	✓ RESOLVED
18	INFO	Typographical errors	✓ RESOLVED

2 Findings

2.1 Global Issues

The issues in this section apply across the whole protocol.

2.1.1 Issues & Recommendations

Issue #01	**`MerkleDistributor` and `OptimisticDistributor` do not support tokens with a fee on transfer**
Severity	INFORMATIONAL
Description	`MerkleDistributor` and `OptimisticDistributor` do not support more tokens with a fee on transfers. This is because the distributors would receive less tokens than they requested. It would also not work with any token which decreases balance of the the Distributors passively.
Recommendation	Consider whether such tokens need to be supported. If so, Paladin will guide the client how to support them.
Resolution	ACKNOWLEDGED The client has indicated that there is no requirement to support tokens with a fee on transfer.

2.2 OptimisticDistributor

`OptimisticDistributor` is the core contract within the Covenant ecosystem. From a high level, the contract allows sponsors to incentivize arbitrary action from other parties by posting a reward for such actions. Whether the actions were successfully executed or not is then validated off-chain, following the UMIP-160 specification.

More specifically, the `OptimisticDistributor` by Covenant allows for "bribing". It is intended to be used to bribe voters from voting platforms like Curve Finance to pass actions in favor of the sponsor.

Sponsors can deposit a reward token and reward token amount of their choosing through `createReward`, and can include various requirements in this request: the proposal that is being bribed, which outcome is desired, and how voters will be paid out for voting on the outcome. The refund (clawback) rules in case the maximum payout is not distributed to voters (or no voting occured before the expiration timestamp) and the error margin permitted.

Each bribe is done in a few steps:

1. The briber calls `createReward` to deposit bribe tokens and submit the bribe details.
2. Voters have until about `earliestProposalTimestamp` to act upon the bribe and execute the requirements.
3. Validators can propose a distribution plan of the bribe tokens to said proposers, or propose a clawback (refund to the briber) in case no action was taken.
4. A dispute period configurable by the briber (with a minimum of 10 minutes) starts where other validators can dispute the proposal. In case a dispute occurs and is accepted, the system restarts at step 3 where validators can propose a new distribution plan.
5. The proposed distribution plan is executed.

2.2.1 Issues & Recommendations

Issue #02	Users can call `createReward` with any whitelisted price identifier
Severity	MEDIUM SEVERITY
Location	Line 173 `require(_getIdentifierWhitelist().isIdentifierSupported(priceIdentifier), "Identifier not registered");`
Description	The `createReward` function allows the user to specify the price identifier. The price identifier represents which specification should be used to resolve the oracle request. Whenever a bribe is made, the briber can specify the priceIdentifier that will be used. This should always be `COVENANT_V1` at the launch of the protocol as that is the only compliant identifier. However, the protocol presently allows any price identifier that is valid to UMA to be provided.
Recommendation	Consider having a local allowlist that also validates that the price identifier is covenant compliant. A simple configurable mapping suffices.
Resolution	RESOLVED Although this is still possible, the risk is mitigated as rewards can no longer be stolen between different distributions. The covenant frontend will also not display any proposals with a bad identifier.

Issue #03	**Gas grief risk: The `priceDisputed` callback might not be guaranteed to execute which could permanently lock-in funds**
Severity	MEDIUM SEVERITY

Description	`OptimisticOracle::459`
	`try OptimisticRequester(requester).priceDisputed(identifier, timestamp, ancillaryData, refund) {} catch {}`
	
	The Optimistic Oracle uses a try-catch structure to call the `OptimisticDistributor`. However, such a structure is potentially vulnerable to gas griefing: https://medium.com/@wighawag/ethereum-the-concept-of-gas-and-its-dangers-28d0eb809bb2
	
	If this structure is vulnerable or ever becomes vulnerable to gas griefing for the `OptimisticDistributor`, it would mean that someone can provide a very specific amount of gas to the `disputePriceFor` call and cause this subcall to run out-of-gas. This would cause the internal state of the `OptimisticDistributor` to become fundamentally out of sync with the Optimistic Oracle and would permanently lock funds in the distributor.
	
	Paladin has carefully tried to execute this grief with the test suite provided by Covenant. As the `priceDisputed` function seems to use sufficiently low gas, we were unable to execute the gas grief as either the `disputePriceFor` ran out-of-gas or succeeded. However, this is no guarantee that gas griefing is impossible.

Recommendation	Consider not relying on the callbacks by the optimistic oracle. This flaw is not a flaw of the Covenant system, but a flaw of the UMA protocol's callbacks.
Resolution	RESOLVED The client moved away from callback logic.

Issue #04	**`ipfsHash` and `customAncillaryData` are private**
Severity	LOW SEVERITY

Location	Line 47
	`bytes customAncillaryData;`
	
	Line 55
	`string ipfsHash;`

Description	The `ipfsHash` parameter of the Proposal struct is not visible anywhere either on-chain or off-chain. Even though proposals are public, any variable-length field will be dropped from the publicly exposed function. The same is true for the `customAncillaryData` bytes from the Reward struct.
Recommendation	Consider marking the above variables as public.
Resolution	RESOLVED The client has confirmed this is not an issue within their version of Solidity.

Issue #05	**There might be a race condition between `increaseReward` and `proposeDistribution`**
Severity	LOW SEVERITY
Description	The contract allows for anyone to increase the allocated rewards until the `earliestProposalTimestamp` has been reached. However, as soon as this timestamp has been reached, validators may propose distributions. This means that there can theoretically be exactly one second between the last increase and the first proposal meaning that it is likely that the first proposal never saw that last increase in this scenario.
Recommendation	Consider no longer allowing increases a few minutes before the `earliestProposalTimestamp`.
Resolution	ACKNOWLEDGED The client has indicated that given this risk is limited only to the proposer whose proposal bond would get slashed through valid dispute, thus they are okay with this issue. After a dispute, anyone can propose a new correct payout. There are no incentives for proposer to rush proposing a payout and it is expected that proposal script would be run only when `earliestProposalTimestamp` would be reached allowing for safe margin to account for block reorg risks.

Issue #06	Unused import: `StoreInterface.sol`
Severity	INFORMATIONAL

Location	Line 14
	`import "../../oracle/interfaces/StoreInterface.sol";`
	
	Line 423-425
	`function _getStore() internal view returns (StoreInterface) { return StoreInterface(finder.getImplementationAddress(OracleInterfaces.Store)); }`

Description	Files and functions that are imported in a contract but not used within said contract could confuse third-party auditors. They also increase the contract unnecessarily.
Recommendation	Consider removing the unused import and function to keep the contract short and simple.
Resolution	RESOLVED

Issue #07	bondToken can be made immutable
Severity	INFORMATIONAL
Description	Variables that are only set in the constructor but never modified can be indicated as such with the immutable keyword. This is considered best practice since it makes the code more accessible for third-party reviewers and saves gas.
Recommendation	Consider making the variable explicitly immutable.
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:
	
	Line 165
	`earliestProposalTimestamp`
	
	In case the IPFS hashes in the ancillary data are only retained for a certain period of time, this timestamp should likely be validated to be before that expiration. If the client would rather leave this unvalidated, that is acceptable as well.
	
	Line 166
	`uint256 optimisticOracleProposerBond`
	
	In case the bond size is unbound, no validator might be able to refute a proposal on short notice because they simply do not have so many bond tokens. It might make sense to cap this parameter. This is especially a risk where voters execute a bribe and the briber then proposes a refund.

Recommendation	Consider validating the function parameters mentioned above.
Resolution	RESOLVED

Issue #09	Typographical errors
Severity	INFORMATIONAL

Description	The contract contains a number of typographical errors which we have consolidated below in a single issue in an effort to keep the report size reasonable.
	
	Line 176
	`require(optimisticOracleLivenessTime < MAXIMUM_LIVENESS, "OO liveness too large");`
	
	This check should be <=.
	
	Line 372
	`function setMerkleDistributor(MerkleDistributor _merkleDistributor) external nonReentrant() {`
	
	Anyone can initialize the `setMerkeDistributor` while it is quite trivial to add some `onlyDeployer` modifier to it.
	
	Line 397
	`* @dev Only accessable as callback through OptimisticOracle on disputes.`
	
	This comment should state *accessible*, not *accessable*.
	
	Line 455
	`function _getProposalId`
	
	This function can be marked as pure.

Recommendation	Consider fixing the typographical errors.
Resolution	RESOLVED

Issue #10	**An unprivileged `syncUmaEcosystemParams` function might be undesirable**
Severity	INFORMATIONAL
Description	An unprivileged `syncUmaEcosystemParams` function might not be desired as an exploiter can then go to all the OpDistributors and sync the newest version and exploit it if an `OptimisticOracle` is deployed with a bug,.
Recommendation	Consider whether this is a risk. If so, consider adding privileges to the sync function.
Resolution	RESOLVED The client has indicated that this is not considered a threat for them as they will fully validate all updates.

Issue #11	Checks-effects-interactions is not adhered to within various functions
Severity	INFORMATIONAL

Description	Various functions do not adhere to the Solidity best practice of checks-effects-interactions.
	
	Line 179
	`rewardToken.safeTransferFrom(msg.sender, address(this), maximumRewardAmount);`
	
	This transfer should be made at the end of the createReward function.
	
	Line 220
	`rewards[rewardIndex].rewardToken.safeTransferFrom(msg.sender, address(this), additionalRewardAmount);`
	
	This transfer should be made at the end of `increaseReward`.
	
	The proposal storing in `proposeDistribution` should arguably be moved upwards but as the functions are properly locked, we do agree that the current ordering within that function is desirable.
	
	This issue has been marked as informational due to all functions being safeguarded with reentrancy guards. It is still pointed out as sometimes reentrancy guards do not suffice as external code execution could do more damage to a system than just reentrancy in the same contract. For example, the exploiter could call the DVM or OO at line 179/220 as these are not reentrancy locked. Still, we could not find an exploit if a malicious party was to do this. In line of safety it is still recommended to patch up any vector that gives an exploiter unnecessary freedom.

Recommendation	Consider fixing the above infringements against the checks-effects-interactions pattern.
Resolution	RESOLVED

2.3 MerkleDistributor

`MerkleDistributor` is a contract that allows an owner to distribute any reward ERC20 token to claimants according to pre-computed Merkle Roots. The owner can specify multiple Merkle Roots distributions with customized reward currencies.

A Merkle Root distribution is valuable as it allows a project to distribute tokens to a large number of users without having to register each user individually on-chain. Instead, it registers a single "checksum" which each user can then prove that they are included in using off-chain data.

A note about this contract is that it was a design decision to keep the code as simple as possible and not include multiple safeguards across the contract, leaving the owner of the contract to validate and correctly create the Merkle Proofs. In case the owner acts maliciously or simply unexpectedly, various functionalities across the contract will fail. Within the larger Covenant system, it is the job of the optimistic oracle to validate that the Merkle trees are correct.

2.3.1 Privileges

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

- `setWindow`
- `deleteWindow`
- `withdrawRewards`
- `transferOwnership`
- `renounceOwnership`

2.3.2 Issues & Recommendations

Issue #12	**The MerkleDistributor unnecessarily allows for separate rewards to be stolen from each other if a single merkle tree allows to withdraw more rewards than were provided to that tree which allows for multiple exploit vectors within the Covenant system**
Severity	HIGH SEVERITY
Description	One of the fundamental concepts of the Covenant system is that the person who requests a bribe can set various parameters to decrease the likelihood that their request will be wrongly or maliciously processed: they can increase the duration that a proposal can be refuted and they can increase the bond value for proposals. This is great as long as an important assumption is satisfied: a bad proposal only affects the briber (sponsor) negatively, and not other sponsors. This assumption is violated within the `MerkleDistributor` contract: all funds from all the different bribers are pooled together in this single contract. As there is no validation on the proposed Merkle trees their total amounts, one bad merkle tree can drain the token balance of the `MerkleDistributor`, stealing all relevant tokens of other bribers.

Proof of Concept:

1. Alice, a large briber, puts up $1 million USDC as a bribe to make a major influence in the Curve Gauge distribution.
2. Bob, an exploiter wants to steal this and creates a second bribe of $1 with difficult to validate parameters: 10 minutes of liveness (shortest liveness allowed) and a large bond amount (eg. $100,000).
3. Bob immediately proposes a Merkle distribution that in fact distributes $1 million USDC to their own wallet.
4. Due to the difficult parameterization of the bribe, no validator might have time to come up with the bond amount in time to refute this proposal. Bob could furthermore DoS the complete chain for 10 minutes to guarantee it.
5. After the 10 minutes expire, Bob steals the $1 million USDC sponsored by Alice.

Being able to potentially steal tokens from other proposals is completely unnecessary and is an absolute target for exploiters. There is absolutely no valid reason to support this.

Recommendation Consider adding a `remainingAmount` parameter to each window that is decreased with each claim. Claims that cause it to underflow should revert.

Resolution RESOLVED

The client has implemented the recommendation preventing spillovers between the different distributions.

Issue #13	**Governance risk: Governance can pre-configure windows with malicious Merkle trees before transferring ownership**
Severity	LOW SEVERITY
Description	The owner of the contract can propose new Merkle trees and drain the contract. Even if the previous issue is resolved, the owner could create a valid Merkle tree and then drain the contract before transferring ownership to the `OptimisticDistributor`. In this case, a Merkle tree would be present with an allowance to drain tokens supplied through the `OptimisticDistributor`.
Recommendation	Consider not using `Ownable` and instead simply deploying with the `OptimisticDistributor` set as an immutable variable.
Resolution	RESOLVED Even though `withdrawRewards` is still present, the `merkledistributor` is now directly deployed by the distributor.

Issue #14	**`ipfsHash` is private**
Severity	LOW SEVERITY
Description	The `ipfsHash` parameter of the `Window` struct is not visible anywhere either on-chain or off-chain. Even though `merkleWindows` is public, any variable-length field will be dropped from the publicly exposed function.
Recommendation	Consider marking the variable as public.
Resolution	RESOLVED The client has validated that this issue is not present within their version of Solidity.

Issue #15	`deleteWindow` lacks validation
Severity	INFORMATIONAL
Description	The contract allows for the owner to delete windows (saved merkle trees). However, the contract lacks validation to check that the window was added and also lacks validation to check if it was already deleted.
Recommendation	Consider adding a `deleted` boolean to the windows (`paused` would be better in our opinion).
Resolution	ACKNOWLEDGED The client has indicated that this method is not relevant to their deployment (which we agree) as their distributor cannot call it, so they will refrain from making modifications to it.

Issue #16	Gas optimizations
Severity	INFORMATIONAL

Description	Throughout the contract, we have identified various sections of the code that can be adjusted for better gas optimization:
	
	Lines 107-112 (example)
	`function setWindow(`
	`uint256 rewardsToDeposit,`
	`address rewardToken,`
	`bytes32 merkleRoot,`
	`string memory ipfsHash`
	`) external onlyOwner {`
	
	Parameters that are sent as `memory` and that are immutable throughout the function body can be declared as `calldata` to avoid unnecessary gas consumption.
	
	`claim` function can be declared as external.
	
	Apart from being a best practice when the function is not used within the contract, this can lead to a lower gas usage in certain cases.
	
	Line 153 (example)
	`uint256 batchedAmount = 0;`
	
	Initializing variables with the default value at declaration consumes unnecessary gas.

Recommendation	Consider implementing the above recommendations to optimize gas usage.
Resolution	RESOLVED It should be noted that a parameter can still be made `calldata` within `claimMulti`.

Issue #17	Unnecessary import: `SafeMath.sol`
Severity	INFORMATIONAL
Description	The Solidity version chosen for this contract is 0.8.0, which automatically includes SafeMath in it, making the SafeMath usage unnecessary.
Recommendation	Consider removing the SafeMath import to keep the contract short and simple.
Resolution	RESOLVED

Issue #18	Typographical errors
Severity	INFORMATIONAL

Description	The contract contains a number of typographical errors which we have consolidated below in a single issue in an effort to keep the report size reasonable.
	
	Line 136
	`IERC20(rewardCurrency).safeTransfer(msg.sender, amount);`
	
	The `rewardCurrency` variable can be declared as `IERC20`.
	
	Line 163
	`address currentRewardToken = address(merkleWindows[_claim.windowIndex].rewardToken);`
	
	The `currentRewardToken` variable can be declared as `IERC20`.

Recommendation	Consider fixing the typographical errors.
Resolution	RESOLVED


PALADIN
BLOCKCHAIN SECURITY