PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Preliminary Report

For Third Web

27 November 2023

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 the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. Paladin is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. Paladin is furthermore allowed to claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for Third Web 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	Third Web
URL	https://thirdweb.com/
Network	Polygon
Language	Solidity
Preliminary	https://github.com/thirdweb-dev/contracts/blob/0370cace93771c0df1db58c1be7db2d0d6d68640/contracts/prebuilts/unaudited/airdrop/AirdropERC20Claimable.sol https://github.com/thirdweb-dev/contracts/blob/0370cace93771c0df1db58c1be7db2d0d6d68640/contracts/lib/MerkleProof.sol
Resolution	

1.2 Contracts Assessed

Name	Contract	Live Code Match
`AidropERC20Claimable`		
`MerkleProof`		

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
Governance	1			
High	1			
Medium	0	-	-	-
Low	0	-	-	-
Informational	5			
Total	**7**	**0**	**0**	**0**

Classification of Issues

Severity	Description
Governance	Issues under this category are where the governance or owners of the protocol have certain privileges that users need to be aware of, some of which can result in the loss of user funds if the governance's private keys are lost or if they turn malicious, for example.
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 AirdropERC20Claimable

ID	Severity	Summary	Status
01	GOV	Governance: `TrustedForwarder` can execute claims on behalf of other addresses	
02	HIGH	Malicious users can steal the entire balance of the contract	
03	INFO	Gas optimizations	
04	INFO	Lack of validation for `expirationTimestamp`	
05	INFO	Lack of `safeTransfer` usage within `_transferClaimedTokens`	

1.3.2 MerkleProof

ID	Severity	Summary	Status
06	INFO	Insufficient `NatSpec` for index determination	
07	INFO	Gas optimizations	

2 Findings

2.1 AirdropERC20Claimable

`AirdropERC20Claimable` is a robust and secure solution designed for the efficient distribution of ERC20 tokens in an airdrop scenario. It utilizes a Merkle Tree-based approach to manage whitelisted participants, ensuring a fair and transparent token claim process.

Key Features:

- **Merkle Proof Verification**: The contract leverages the `MerkleProof` library to authenticate claims. This feature allows it to verify whether a participant is whitelisted and entitled to claim tokens up to a specified amount, which is provided by the user as a parameter to the claim function `_proofMaxQuantityForWallet`.

- **Whitelisted Claims:** Participants whose addresses are included in the Merkle Tree can claim tokens up to their pre-assigned amounts. This is validated through a cryptographic proof that confirms their inclusion in the whitelist.

- **Open Claims:** Non-whitelisted participants are also eligible to claim tokens but are subject to an `openClaimLimitPerWallet` limitation, which caps the amount they can receive.

Additionally, the contract's `claim` function is publicly accessible, allowing any user (whitelisted or not) to initiate a token claim. Tokens are allocated for the airdrop are also held in a separate address. Upon a successful claim, the contract triggers a transfer of the tokens to the receiver address, ensuring a secure and direct distribution process.

To facilitate interactions, the `Multicall` library is inherited which allows multiple claims to a different recipient address in the same block, and the `ERC2771Context` library has been inherited to allow the addition of trusted forwarders.

2.1.1 Issues & Recommendations

Issue #01	**Governance: `TrustedForwarder` can execute claims on behalf of other addresses**
Severity	GOVERNANCE
Description	Upon contract deployment, one or more `trustedForwarders` can be determined. These addresses can pad any address towards a function call, which then will be used as `_msgSender`. This ultimately allows any trusted forwarder to execute claims on behalf of any address. Moreover, the contract is desired to be used as an implementation for a proxy.
Recommendation	Consider being very careful with the selection of trusted forwarders, as they can essentially consume the allowance from all addresses.
Resolution	

Issue #02	Malicious users can steal the entire balance of the contract
Severity	HIGH SEVERITY
Description	The `AidropERC20Claimable` contract allows users to claim tokens. The maximum amount a user can claim is given by the `_proofMaxQuantityForWallet` of the user's leaf. This amount is proved by the Merkle Tree and prevents users from claiming more than what they were allocated. Users that were not added to the Merkle Tree can still claim tokens, however, the amount they can claim is determined by `openClaimLimitPerWallet` However, a Sybil attack can be performed by malicious users to claim via `openClaimLimitPerWallet` multiple times and up to the entire balance of the contract. This could even prevent users that were added to the Merkle Tree to claim any tokens.
Recommendation	Consider adding a timestamp from where users can claim the `openClaimLimitPerWallet` amount. Note that from this point, the remaining balance could be stolen by only one user that creates multiple addresses to claim everything if they are fast enough. The easiest way to fix this issue is to simply not let anyone claim the airdrop and distribute the remaining balance, if any, through different methods such as another airdrop, rewards to a farm, etc.
Resolution	

Issue #03	Gas optimizations
Severity	INFORMATIONAL

Description	L41 - 57
	`/// @dev address of token being airdropped.`
	`address public airdropTokenAddress;`
	
	`/// @dev address of owner of tokens being airdropped.`
	`address public tokenOwner;`
	
	`[...]`
	
	`/// @dev airdrop expiration timestamp.`
	`uint256 public expirationTimestamp;`
	
	`/// @dev claim limit for open/public claiming without allowlist.`
	`uint256 public openClaimLimitPerWallet;`
	
	`/// @dev merkle root of the allowlist of addresses eligible to claim.`
	`bytes32 public merkleRoot;`
	
	Those values could be made immutable to save some gas.
	
	Note that this is true only if this is the first time this contract is deployed, otherwise this could create storage collisions as it is an upgradeable proxy,.

Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	

Issue #04	Lack of validation for `expirationTimestamp`
Severity	INFORMATIONAL
Description	This variable should be properly validated to either be zero or in the future, otherwise claims will revert immediately after deployment.
Recommendation	Consider validating the `expirationTimestamp` accordingly.
Resolution	

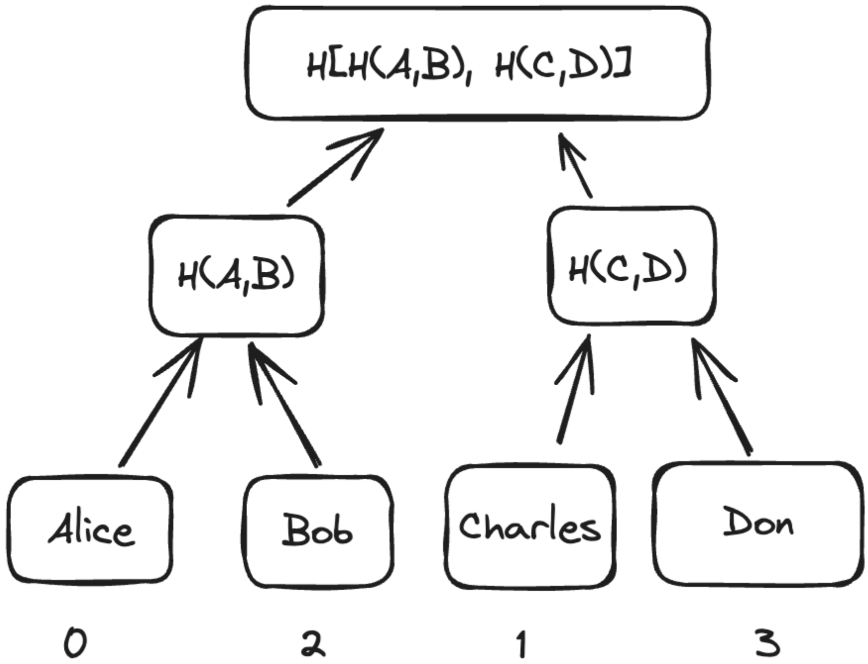
Issue #05	Lack of `safeTransfer` usage within `_transferClaimedTokens`
Severity	INFORMATIONAL
Location	L164 `require(IERC20(airdropTokenAddress).transferFrom(tokenOwner, _to, _quantityBeingClaimed), "transfer failed");`
Description	Within the `_transferClaimedTokens` function, `transferFrom` is used to transfer tokens from `tokenOwner` to the recipient `_to` . This will not work for tokens that returns `false` on transfer (or malformed tokens that do not have a return value).
Recommendation	Consider using `safeTransfer` instead of `transferFrom` .
Resolution	

2.2 MerkleProof

`MerkleProof` is a customized cryptography contract developed by the ThirdWeb team to correctly recreate a merkle root based on a leaf and the corresponding proofs.

It explicitly returns if the recreated merkle root is the same as the provided input root. It is important to mention that the the root creation is based on hashing from left to right, which perfectly aligns with OpenZeppelin's logic. However, it is still important to keep this in mind when creating the merkle tree. Additionally, the index of the corresponding leaf is returned.

2.2.1 Issues & Recommendations

Issue #06	Insufficient `NatSpec` for index determination
Severity	INFORMATIONAL
Description	The index determination for the specific leafs id is as follows:  Unfortunately, we could not explore the idea behind this outcome.
Recommendation	Consider implementing proper documentation for the id creation.
Resolution	

Issue #07	Gas optimization
Severity	INFORMATIONAL

Description	The merkle tree recreation is based on a left to right hashing, which means if a < b then `hash(a,b)`, otherwise `hash(b,a)`.
	
	This is expressed as follows:
	
	`if (computedHash <= proofElement) {`
	`    // Hash(current computed hash + current element of the proof)`
	`    computedHash = keccak256(abi.encodePacked(computedHash, proofElement));`
	`} else {`
	`    // Hash(current element of the proof + current computed hash)`
	`    computedHash = keccak256(abi.encodePacked(proofElement, computedHash));`
	`    index += 1;`
	`}`
	
	As can be seen from above, in the first `if` check, the equal sign is included. This is an unnecessary check and wastes gas as it can be implicitly included in the `else` clause since for that specific scenario, it does not make a difference whether it is `hash(a,b)` or `hash(b,a)` as the outcome will be exactly the same.
	
	It is important that this change might have an impact on the index, however, since the index is a) not actively used and b) not described properly, this impact might be negligible.
	
	Moreover, the hashing mechanism can be implemented in a more gas-friendly manner. An example can be found in OpenZeppelin's MerkleRoot contract:
	
	`function _efficientHash(bytes32 a, bytes32 b) private pure returns (bytes32 value) {`
	`    /// @solidity memory-safe-assembly`
	`    assembly {`
	`        mstore(0x00, a)`
	`        mstore(0x20, b)`
	`        value := keccak256(0x00, 0x40)`
	`    }`
	`}`

Recommendation	Consider removing the equal check to save gas and changing the hashing mechanism to the methodology above.
Resolution	

PALADIN
BLOCKCHAIN SECURITY