// Security Assessment

10.23.2024 - 11.01.2024

StakeWeight, StakingRewardDistributor & Airdrop

Reown (fka WalletConnect)

HALBORN

StakeWeight, StakingRewardDistributor & Airdrop - Reown (fka WalletConnect)

Prepared by: HALBORN

Last Updated Unknown date

Date of Engagement: October 23rd, 2024 - November 1st, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
5	0	0	0	2	3

TABLE OF CONTENTS

1. Introduction

`WalletConnect` engaged `Halborn` to conduct a security assessment on their **Solidity** smart contracts beginning on October 23rd, 2024 and ending on November 1st, 2024. The security assessment was scoped to the smart contracts provided in the **WalletConnectFoundation-Contracts GitHub repository**, commit hashes and further details can be found in the Scope section of this report.

`WalletConnect` is developing a decentralized infrastructure for permissionless, interoperable messaging between dApps and wallets. This project aims to gradually transition from a permissioned to a fully permissionless network, starting with the decentralization of the storage layer.

2. Assessment Summary

The team at Halborn assigned one full-time security engineer to check the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing and smart-contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functionality operates as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which have been partially addressed by the `WalletConnect team`. The main ones were the following:

- `Implement a two-step ownership transfer mechanism to ensure ownership transfers are confirmed, reducing the risk of accidental or unauthorized reassignments.`
- `Add a validation in the injectReward function to prevent reward injections with timestamps prior to startWeekCursor to ensure all injected rewards are properly distributed.`
- `Introduce an emergencyWithdraw function to allow users to withdraw their staked funds under specific emergency conditions when the contract is paused.`
- `Apply the nonReentrant modifier to fund-transfer functions and ensure state changes occur before transfers to prevent reentrancy risks, even if they are protected by onlyOwner.`
- `Add pre-transfer checks for allowance and balance in the claimTokens function to prevent failed transactions and save gas by avoiding unnecessary operations before transfers.`

3. Test Approach And Methodology

`Halborn` performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could lead to arithmetic related vulnerabilities.
- Manual testing by custom scripts.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Static Analysis of security for scoped contract, and imported functions. (`Slither`).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: contracts

(b) Assessed Commit ID: 74de69f

(c) Items in scope:

- StakeWeight.sol
- StakingRewardDistributor.sol
- Airdrop.sol

Out-of-Scope: Third party dependencies and economic attacks.

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	2	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
SINGLE-STEP OWNERSHIP TRANSFER RISK	LOW	PARTIALLY SOLVED - 11/06/2024
RETROACTIVE REWARD INJECTION	LOW	SOLVED - 11/06/2024
ABSENCE OF EMERGENCY WITHDRAWAL MECHANISM FOR USERS	INFORMATIONAL	PARTIALLY SOLVED - 11/06/2024
LACK OF NON-REENTRANCY PROTECTION IN FUND-TRANSFER FUNCTIONS	INFORMATIONAL	SOLVED - 11/06/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING ALLOWANCE AND BALANCE CHECKS BEFORE TOKEN TRANSFER	INFORMATIONAL	ACKNOWLEDGED - 11/06/2024

7. FINDINGS & TECH DETAILS

7.1 SINGLE-STEP OWNERSHIP TRANSFER RISK

// LOW

Description

The `Airdrop` contract utilizes OpenZeppelin's `AccessControl` to manage roles and permissions. While `AccessControl` provides a basic structure for role management, it lacks advanced security features such as time delays or multi-step confirmations for changes to critical roles, like the `DEFAULT_ADMIN_ROLE`.

```
7  import { ReentrancyGuard } from "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
8  import { AccessControl } from "@openzeppelin/contracts/access/AccessControl.sol";
9  import { MerkleProof } from "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";
10 import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
11
12 /// @title Airdrop
13 /// @notice A contract for distributing tokens via a Merkle airdrop
14 /// @author WalletConnect
15 contract Airdrop is AccessControl, Pausable, ReentrancyGuard {
16     using SafeERC20 for IERC20;
```

Additionally, the `OwnableUpgradeable` module, used in both the `StakeWeight` and `StakingRewardDistributor` contracts, implements a single-step ownership transfer process where ownership is instantly reassigned to a new address via `transferOwnership(newOwner)`. This single-step approach presents a security risk, as any error or oversight in specifying the `newOwner` could irreversibly transfer control to an unintended address, exposing the contract to potential misuse.

```
6  import { Math } from "@openzeppelin/contracts/utils/math/Math.sol";
7  import { OwnableUpgradeable } from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol"
8  import { ReentrancyGuardUpgradeable } from "@openzeppelin/contracts-upgradeable/utils/ReentrancyGuardU
9  import { SafeCast } from "@openzeppelin/contracts/utils/math/SafeCast.sol";
10 import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
11 import { StorageSlot } from "@openzeppelin/contracts/utils/StorageSlot.sol";
12
13 import { Pauser } from "./Pauser.sol";
14 import { WalletConnectConfig } from "./WalletConnectConfig.sol";
15
16 /**
17  * @title StakeWeight
18  * @notice This contract implements a vote-escrowed token model for WCT (WalletConnect Token)
19  * to create a staking mechanism with time-weighted power.
20  * @dev This contract was inspired by Curve's veCRV and PancakeSwap's veCake implementations.
21  * @author WalletConnect
22  */
23 contract StakeWeight is Initializable, OwnableUpgradeable, ReentrancyGuardUpgradeable
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (2.0)

Recommendation

It is recommended to consider using `AccessControlDefaultAdminRules` instead of `AccessControl` for the `Airdrop` contract to enhance security for changes to the `DEFAULT_ADMIN_ROLE`. This module enables features such as time delays or requiring multiple confirmations before finalizing role changes.

Additionally, for the `StakeWeight` and `StakeRewardDistributor` contracts, it is advisable to implement a two-step ownership transfer process, such as OpenZeppelin's `Ownable2StepUpgradeable`. This approach requires the new owner to confirm the transfer, reducing the risk of accidental or unauthorized reassignments.

Remediation Comment

PARTIALLY SOLVED: The **WalletConnect team** has partially solved the issue.

The `StakingRewardDistributor` contract has been updated to use OpenZeppelin's `Ownable2StepUpgradeable` in commit **187cc15**.

However, the `StakeWeight` contract has been switched from `OwnableUpgradeable` to the `AccessControlUpgradeable` library in commit **c289e88**, while the `Airdrop` contract continues to use `AccessControl`.

The client stated the following:
"*All these contracts would have the Owner / Default Admin Role being Timelocks, so we don't want to have those accumulating and those reduce the risk of sudden access-control changes.*"

7.2 RETROACTIVE REWARD INJECTION

// LOW

Description

The `injectReward` function in the `StakingRewardDistributor` contract allows rewards to be injected for any specified timestamp, including past weeks relative to the global `startWeekCursor` parameter, which marks the starting timestamp for rewards distribution.

If rewards are injected with a timestamp before `startWeekCursor`, they will not be distributed to users, as any tokens injected with an outdated timestamp are effectively ignored in the distribution process.

```
532 function injectReward(uint256 timestamp, uint256 amount) external onlyOwner {
533     _injectReward(timestamp, amount);
534 }
535
536 /// @notice Inject rewardToken for currect week into the contract
537 /// @param amount The amount of rewardToken to be distributed
538 function injectRewardForCurrentWeek(uint256 amount) external onlyOwner {
539     _injectReward(block.timestamp, amount);
540 }
541
542 function _injectReward(uint256 timestamp, uint256 amount) internal {
543     IERC20(config.getL2wct()).safeTransferFrom(msg.sender, address(this), amount);
544     lastTokenBalance += amount;
545     totalDistributed += amount;
546     uint256 weekTimestamp = _timestampToFloorWeek(timestamp);
547     tokensPerWeek[weekTimestamp] += amount;
548 }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:C (2.0)

Recommendation

It is recommended to update the `injectReward` function to check that rewards are injected only for timestamps strictly after `startWeekCursor`.

Remediation Comment

SOLVED: The **WalletConnect team** has solved this issue by checking the timestamp.

7.3 ABSENCE OF EMERGENCY WITHDRAWAL MECHANISM FOR USERS

// INFORMATIONAL

Description

The `StakeWeight` contract lacks an emergency withdrawal function, which prevents users from recovering their staked assets under emergency conditions.

Although the pause functionality halts contract activity, it does not provide users with direct access to their locked tokens during critical failures or unforeseen contract issues.

BVSS

AO:S/AC:L/AX:H/R:N/S:U/C:N/A:C/I:N/D:C/Y:N (0.8)

Recommendation

It is recommended to implement an `emergencyWithdraw` function that allows users to withdraw their staked funds without penalties if the contract remains paused for a certain duration or during an emergency.

Ensure this function can only be triggered by a privileged role and under predefined conditions, such as when the contract is actively paused.

Remediation Comment

PARTIALLY SOLVED: The **WalletConnect team** has partially solved the issue by adding the `forceWithdrawAll` function to the **StakeWeight** contract.
They stated the following: "*We might implement this in the future in a proxy upgrade, in the meantime, we have this method*".

7.4 LACK OF NON-REENTRANCY PROTECTION IN FUND-TRANSFER FUNCTIONS

// INFORMATIONAL

Description

The functions `kill`, `injectReward`, and `injectRewardForCurrentWeek` in the `StakingRewardDistributor` contract perform direct fund transfers but lack the `nonReentrant` modifier, which is critical for reentrancy protection.

While these functions are restricted by the `onlyOwner` modifier, they remain at risk if interactions with the contract are compromised. If the **ERC20** contract is malicious or has been compromised, it could execute additional code during the transfer, potentially attempting recursive calls to these functions before they finish updating the contract state.

```
532 function injectReward(uint256 timestamp, uint256 amount) external onlyOwner {
533     _injectReward(timestamp, amount);
534 }
535
536 /// @notice Inject rewardToken for currect week into the contract
537 /// @param amount The amount of rewardToken to be distributed
538 function injectRewardForCurrentWeek(uint256 amount) external onlyOwner {
539     _injectReward(block.timestamp, amount);
540 }
541
542 function _injectReward(uint256 timestamp, uint256 amount) internal {
543     IERC20(config.getL2wct()).safeTransferFrom(msg.sender, address(this), amount);
544     lastTokenBalance += amount;
545     totalDistributed += amount;
546     uint256 weekTimestamp = _timestampToFloorWeek(timestamp);
547     tokensPerWeek[weekTimestamp] += amount;
548 }
```

Additionally, some of these functions modify the contract's state ***after*** initiating the transfer, increasing the potential impact of reentrancy attacks.

BVSS

AO:S/AC:L/AX:H/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (0.3)

Recommendation

Add the `nonReentrant` modifier to `kill`, `injectReward`, and `injectRewardForCurrentWeek` to prevent reentrancy risks. Additionally, ensure that all state changes occur before any fund transfer to minimize vulnerabilities.

Remediation Comment

SOLVED: The **WalletConnect team** has solved this issue by adding the `nonReentrant` modifier to the mentioned functions.

7.5 MISSING ALLOWANCE AND BALANCE CHECKS BEFORE TOKEN TRANSFER

// INFORMATIONAL

Description

In the `claimTokens` function, the contract uses `safeTransferFrom` without first checking if the `reserveAddress` has given enough allowance or has enough tokens.

```
67  function claimTokens(
68      uint256 index,
69      uint256 amount,
70      bytes32[] calldata merkleProof
71  )
72      external
73      whenNotPaused
74      nonReentrant
75  {
76      if (amount == 0) revert InvalidAmount();
77      if (claimed[msg.sender]) revert AlreadyClaimed();
78
79      claimed[msg.sender] = true;
80
81      bytes32 node = keccak256(abi.encodePacked(index, msg.sender, amount));
82      if (!MerkleProof.verify(merkleProof, merkleRoot, node)) revert InvalidProof();
83
84      token.safeTransferFrom(reserveAddress, msg.sender, amount);
85
86      emit TokensClaimed({ recipient: msg.sender, amount: amount });
87  }
```

If the allowance or balance is insufficient, the transfer will fail, causing the transaction to revert and the user to lose gas fees. This is not a security issue, as the transaction would eventually fail during the transfer. However, it leads to inefficiency because the contract may perform unnecessary operations, such as verifying the Merkle proof, before failing due to lack of allowance or balance.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to add checks at the beginning of the `claimTokens` function to verify that the `reserveAddress` has sufficient allowance and balance. This allows the contract to revert early if the conditions are not met, saving gas for users and avoiding unnecessary Merkle proof checks, making the process more efficient.

Remediation Comment

ACKNOWLEDGED: The **WalletConnect team** has acknowledged this issue, stating: "*This checks will save gas for reverts that get included in blocks, but add gas for successful transactions, so we decide to let it revert with more gas*".

STATIC ANALYSIS REPORT

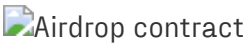
Description

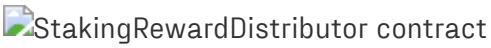
Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

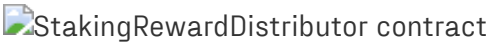
The security team assessed all findings identified by the Slither software, however, findings with related to external dependencies are not included in the below results for the sake of report readability.

Slither Results

The findings obtained as a result of the Slither scan were reviewed, and not included in the report because they were determined as **false positives.**







Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.