SHERLOCK

Security Review For Covalent

Private Best Efforts contest prepared for: **Covalent**
Lead Security Expert: **PUSH0**
Date Audited: **October 25 - October 29, 2024**

Introduction

Covalent is a modular data infrastructure layer that's dedicated to solving long-term data availability on Ethereum. The Covalent Network's Ethereum Wayback Machine (EWM) ensures secure, decentralized access to Ethereum's transaction data after it's purged. This contest focuses on auditing the contracts for the EWM Light Client, a product designed to ensure data availability in the EWM.

Scope

Repository: covalenthq/ewm-lc-contracts

Branch: main

Audited Commit: c94d52563f7e47f7b0c8e2b13beb43a7a50806dc

Final Commit: f282fca0fa8b283984e36fe18a44310decd48eaf

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

High	Medium
0	2

Issues not fixed or acknowledged

High	Medium
0	0

Security experts who found valid issues

jennifer37
PUSH0
sakshamguruji
Atharv
0xmystery
merlin
hunter_w3b
pseudoArtist
Kirkeelee
0xeix

Issue M-1: `EwmNftController` is not strictly compliant with the EIP-4907 standards

Source: https://github.com/sherlock-audit/2024-10-covalent-2-judging/issues/16

The protocol has acknowledged this issue.

Found by

Atharv, PUSH0, jennifer37, sakshamguruji

Summary

Per the contest README:

> The codebase expected to comply with the following EIPs
>
> - **Strictly compliant**:
>   - EIP-4907 (rental standard for ERC721)

Per the EIP-4907 standard:

> The `supportsInterface` method MUST return `true` when called with `0xad092b5c`

However the interface `IERC4907`, inherited by `EwmNftController`, does not adhere to this requirement. Because the `EwmNftController` is a rental NFT that is expected to follow the EIP-4907 standard, this in turn breaks the rental NFT's compliance.

Per the Sherlock rules:

> If a contract is in contest Scope, then all its parent contracts are included by default.

Root Cause

The `interfaceId` of an interface/contract type is determined by XOR-ing all of their function selectors together. The `0xad092b5c` is the magic value that is the `interfaceId` of `IERC4907` defined in the standard:

https://eips.ethereum.org/EIPS/eip-4907

However, the current function signature/selector of `setUser` is as follow:

```
function setUser(uint256 tokenId, address user) external;
```

https://github.com/sherlock-audit/2024-10-covalent-2/blob/main/ewm-lc-contracts/contracts/interfaces/IERC4907.sol#L15

Which is different from the 4907 standard, having an extra `expires` parameter:

```
function setUser(uint256 tokenId, address user, uint64 expires) external;
```

causing the `interfaceId` to become different. Then it is impossible for `supportInterface` to return `true` for the interface ID `0xad092b5c`.

https://github.com/sherlock-audit/2024-10-covalent-2/blob/main/ewm-lc-contracts/contracts/EwmNftController.sol#L211-L213

Internal pre-conditions

None

External pre-conditions

None

Attack Path

N/A

Impact

Non-compliance with the ERC-4907 standard which the protocol expects **strictly**.

Smart contract integration issues may also arise for contracts interacting with the NFT and expecting strict interface support, given that the NFT holders are meant to be EWM Light Client runners, which can be expected to be smart contract users, as opposed to regular app users.

PoC

The following test, when run on Remix IDE, will yield a different value for `type(IERC4907).interfaceId` than the standard. Replacing it with the correct interface will also yield the correct `interfaceId`

```
// SPDX-License-Identifier: CC0-1.0

pragma solidity ^0.8.0;

// copy pasted from contest repo
interface IERC4907 {
  // Logged when the user of a token assigns a new user or updates expires
  /// @notice Emitted when the `user` of an NFT or the `expires` of the `user` is
 ↪  changed
```

```
  /// The zero address for user indicates that there is no user address
  event UpdateUser(uint256 indexed tokenId, address indexed user, uint64 expires);

  /// @notice set the user and expires of a NFT
  /// @dev The zero address indicates there is no user
  /// Throws if `tokenId` is not valid NFT
  /// @param user  The new user of the NFT
  function setUser(uint256 tokenId, address user) external;

  /// @notice Get the user address of an NFT
  /// @dev The zero address indicates that there is no user or the user is expired
  /// @param tokenId The NFT to get the user address for
  /// @return The user address for this NFT
  function userOf(uint256 tokenId) external view returns (address);

  /// @notice Get the user expires of an NFT
  /// @dev The zero value indicates that there is no user
  /// @param tokenId The NFT to get the user expires for
  /// @return The user expires for this NFT
  function userExpires(uint256 tokenId) external view returns (uint64);
}

contract Test {

    function testInterfaceId() public pure returns(bytes4){
        return type(IERC4907).interfaceId; // 0x96745459
    }

}
```

Mitigation

The function `setUser` should include a (possibly empty) `expiry` parameter to match the correct interface.

Discussion

noslav

For clarification, this is fine plus expected, and an agreed upon change (and a small deviation) from the ERC 4907 standard that `setUser` does not have the `expires` argument to the fn call.

This was done purposely in a newer version of the deployed contracts since we wanted to strictly enforce the expiry dates using the `expiryRanges` stored previously on the `EwmNftController.sol` contract. The user should not be able to overwrite this in any case since the contract enforces a usage rights contract to light clients and their respective operational rewards for a limited pre-fixed (fixed expiry) period using this model.

Issue M-2: Improper Limit Enforcement Allows Attacker to Obtain More Than 10 NFTs Per Address

Source: https://github.com/sherlock-audit/2024-10-covalent-2-judging/issues/44

The protocol has acknowledged this issue.

Found by

0xeix, 0xmystery, Kirkeelee, PUSH0, hunter_w3b, jennifer37, merlin, pseudoArtist, sakshamguruji

Summary

The contest README states that, in integer mode, there is a maximum of 10 NFTs per address.

> Maximum 10 NFTs per address in integer mode
> Integer-based allocation limits (max 10 NFTs)

However, in the code implementation, users can bypass the intended 10 NFT per address limit when `isIntegerAllowance` is set to `true`. This occurs because the limit is only checked during each individual allocation call, not cumulatively across multiple calls. A user can make multiple allocation calls, each under the 10 NFT limit, to acquire more than 10 NFTs in total.

Root Cause

The insufficient enforcement of the 10 NFT per address limit stems from a flaw in the contract's logic. The check for the limit (`require(nftQuantity<=10,'Canonlystake10NFTsatmaxinintegerallowance');`) is performed only within the `_allocate` function and only considers the number of NFTs requested in a single call, not the total number of NFTs already allocated to that address.

```
function _allocate(uint256 nftQuantity) internal {
  uint256 stakeAmount = nftQuantity * stakePrice;
  require(
    totalStakeReceived + stakeAmount <= maxTotalStakeable,
    'Exceeds max total allowable stake'
  );

  if (isIntegerAllowance) {
```

```
@>      require(nftQuantity <= 10, 'Can only stake 10 NFTs at max in integer
↪  allowance');
    }

    _transferToContract(_msgSender(), stakeAmount);
    totalStakeReceived += stakeAmount;
    nftMintAllowance[_msgSender()] += nftQuantity;
    totalNftToMint[_msgSender()] += nftQuantity;
    userStakeAmount[_msgSender()] += stakeAmount;

    emit Allocation(_msgSender(), nftQuantity);
  }
```

https://github.com/sherlock-audit/2024-10-covalent-2/blob/main/ewm-lc-contracts/contracts/EwmNftAllowance.sol#L91-L109

Internal pre-conditions

No response

External pre-conditions

No response

Attack Path

1. The contract admin sets `isIntegerAllowance` to `true`.
2. An attacker repeatedly calls the `whitelistedAllocation` function, each time requesting a number of NFTs upto 10.
3. The attacker successfully accumulates more than 10 NFTs, bypassing the intended limit.

Impact

This bypass undermines the intended allocation mechanism. The contract's stated limitation of 10 NFTs per address is not enforced, potentially leading to unfair distribution and a depletion of the total available NFTs.

PoC

No response

Mitigation

The most effective mitigation is to modify the `_allocate` function to check the total number of NFTs already allocated to an address before allowing a new allocation. This requires tracking the total allocated NFTs for each address, either through a dedicated mapping or by modifying the existing `totalNftToMint` variable to accurately reflect the cumulative allocation, even after `releaseNftStake` is called.

This would require a change to how `releaseNftStake` updates `totalNftToMint`. A robust solution would also ensure that the `totalNftToMint` variable is accurately updated even after the `releaseNftStake` function is executed. This ensures the contract's behavior aligns with the documentation.

Example of Mitigation (Aggregate Allocation):

Replace the `whitelistedAllocation` and `_allocate` functions with something like this:

```
function whitelistedAllocation(uint256 nftQuantity, bytes32[] calldata merkleProof)
↪  public virtual whenNotPaused onlyDuringAllocation nonReentrant {
    require(checkWhitelist(_msgSender(), merkleProof), 'proof invalid');
    _allocate(nftQuantity);
}

function _allocate(uint256 nftQuantity) internal {
    require(nftQuantity > 0, "NFT quantity must be greater than zero");
    if (isIntegerAllowance) {
        require(nftQuantity <= 10, "Can only stake a maximum of 10 NFTs in integer
↪  allowance mode.");
    }
    uint256 currentAllowance = nftMintAllowance[_msgSender()];
    require(currentAllowance + nftQuantity <= 10, "Total allocation exceeds 10
↪  NFTs."); //Check against 10 NFT limit

    uint256 stakeAmount = nftQuantity * stakePrice;
    require(totalStakeReceived + stakeAmount <= maxTotalStakeable, 'Exceeds max
↪  total allowable stake');

    _transferToContract(_msgSender(), stakeAmount);
    totalStakeReceived += stakeAmount;
    nftMintAllowance[_msgSender()] += nftQuantity;
    totalNftToMint[_msgSender()] += nftQuantity;
    userStakeAmount[_msgSender()] += stakeAmount;

    emit Allocation(_msgSender(), nftQuantity);
}
```

This revised code prevents the bypass by ensuring that the total number of NFTs allocated to an address never exceeds 10 when `isIntegerAllowance` is true. The user must request all their NFTs in a single transaction. This is a much more robust solution than the original implementation.

Discussion

noslav

The intended functionality is not broken. The idea is to prevent a single user consuming up all the allocation in a single tx call. They can however make multiple txs from the same account to own more than 10. This gives others participating during the sale to acquire some allocation for themselves. Anyone however can make more `whitelistedAllocation` txs with acquiring 10 nfts allocations during the sale period and acquire to the max limit possible in the sale.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.