prePO Solo Audit by cccz Findings & Analysis Report

2022-09-06

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 Solo Audit is an event where a top Code4rena contributor, commonly referred to as a warden or a team, reviews, audits and analyzes smart contract logic in exchange for a bounty provided by sponsoring projects.

During the Solo Audit outlined in this document, C4 conducted an analysis of the prePO code. The audit took place between August 23-31, 2022.

Following the Solo Audit, warden cccz reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit contest report.

Wardens

Audit and mitigation review completed by cccz.

Final report assembled by itsmetechjay.

Summary

The prePO Solo Audit yielded 4 MEDIUM vulnerabilities. There were also 3 informational findings and 13 gas optimizations reported.

Of these, 3 vulnerabilities and 13 gas optimizations have been fixed. 3 informational findings have been acknowledged.

The codebase in question had already undergone one prior Code4rena contest.

Scope

Code reviewed consisted of the following files:

- PPO.sol
- AccountList.sol
- BlocklistTransferHook.sol
- RestrictedTransferHook.sol

- MiniSales.sol
- AllowlistPurchaseHook.sol
- Vesting.sol
- TokenShop.sol
- PurchaseHook.sol
- Pausable.sol
- SafeOwnable.sol
- SafeOwnableUpgradeable.sol
- SafeOwnableCaller.sol
- SafeAccessControlEnumerable.sol
- SafeAccessControlEnumerableUpgradeable.sol
- SafeAccessControlEnumerableCaller.sol
- WithdrawERC20.sol
- WithdrawERC721.sol
- WithdrawERC1155.sol

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

Medium Risk Findings (4)

[M-01] Token transfers do not verify that the tokens were successfully transferred

Some tokens (like zrx) do not revert the transaction when the transfer/transferfrom fails and return false, which requires us to check the return value after calling the transfer/transferfrom function.

In the purchase functions of MiniSales and TokenShop contracts, if _paymentToken is such a token, the user can purchase _saleToken and NFT without spending any tokens.

In the claim function of the Vesting contract, if _token is such a token, the user may lose his vested tokens.

Proof of Concept

https://github.com/prepo-io/prepo-monorepo/blob/9ed5587e557a1189df6c5bb0229ad8c171eaabce/apps/smart-contracts/token/contracts/mini-sales/MiniSales.sol#L25-L26

https://github.com/prepo-io/prepo-monorepo/blob/9ed5587e557a1189df6c5bb0229ad8c171eaabce/apps/smart-contracts/token/contracts/token-shop/TokenShop.sol#L41-L42

https://github.com/prepo-io/prepo-monorepo/blob/9ed5587e557a1189df6c5bb0229ad8c171eaabce/apps/smart-contracts/token/contracts/vesting/Vesting.sol#L66-L67

Recommended Mitigation Steps

Use SafeERC20's safeTransfer/safeTransferFrom functions

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol

cccz (warden) commented:

> Only security-compliant tokens such as USDC and PPO are currently planned to be used in the contracts.

[M-02] TokenShop: purchase function should add _purchasePrices parameter

In the purchase function of the TokenShop contract, the price of the NFT is represented by _contractToIdToPrice, and _contractToIdToPrice can be set by the owner in the setContractToIdToPrice function.

If setContractToIdToPrice and the purchase function are executed in the same block, the user may suffer a loss due to the new _contractToIdToPrice.

Consider the following scenarios:

The current _contractToIdToPrice is 500. the user likes the price, and the purchase function is called.

But at this time, the setContractToIdToPrice function is called, setting the _contractToIdToPrice to 1000, and this transaction occurs before executing the purchase function, causing the user to execute the purchase function in the case of _contractToIdToPrice 1000.

Proof of Concept

https://github.com/prepo-io/prepo-monorepo/blob/9ed5587e557a1189df6c5bb0229ad8c171eaabce/apps/smart-contracts/token/contracts/token-shop/TokenShop.sol#L35-L41

https://github.com/prepo-io/prepo-monorepo/blob/9ed5587e557a1189df6c5bb0229ad8c171eaabce/apps/smart-contracts/token/contracts/token-shop/TokenShop.sol#L26-L31

Recommended Mitigation Steps

Add the *purchasePrices parameter to the purchase function of TokenShop, and verify that _purchasePrices[i] >= _contractToldToPrice[*tokenContracts[i]][_ids[i]]

prePO (sponsor) confirmed and resolved:

> Fixed in PR 355.

cccz (warden) reviewed mitigation:

> Fixed by adding `purchasePrices` parameter in `purchase()` to avoid race condition and ensure that tokens are not purchased at higher prices than the user intended.

[M-03] SafeAccessControlEnumerableCaller and SafeOwnableCaller contracts lack access control

The SafeAccessControlEnumerableCaller and SafeOwnableCaller contracts are abstract contracts that are used to call privileged functions in the SafeAccessControlEnumerable and SafeOwnable contracts.

Contracts that inherit from *caller are required to override the non-view functions in the *caller contract to prevent anyone from calling privileged functions in *caller.

But once the inheriting contract does not override all the non-view functions, anyone can call the unoverridden privileged functions in the *caller.

Since the functions implemented in the abstract contract are no longer forced to be implemented again by the inheriting contract, the contract can be deployed even if the inheriting contract does not override all the non-view functions.

Proof of Concept

https://github.com/prepo-io/prepo-monorepo/blob/9ed5587e557a1189df6c5bb0229ad8c171eaabce/packages/prepo-shared-contracts/contracts/SafeAccessControlEnumerableCaller.sol#L8-L55

https://github.com/prepo-io/prepo-monorepo/blob/9ed5587e557a1189df6c5bb0229ad8c171eaabce/packages/prep

o-shared-contracts/contracts/SafeOwnableCaller.sol#L7-L23

Recommended Mitigation Steps

Consider adding simple access control to the *caller contract. Or instead of implementing functions in the *caller, provide the code as comments or documentation for use in inheriting contracts.

prePO (sponsor) confirmed and resolved:

> Fixed in PR 352.

cccz (warden) reviewed mitigation:

> Fixed by removing all write methods from `SafeOwnableCaller` and `SafeAccessControlEnumerableCaller` to ensure that the inheriting contracts need to override all write methods. This will avoid missing out of access control on some critical write methods.

[M-04] No storage gap for upgradeable contracts

For SafeAccessControlEnumerableUpgradeable and SafeOwnableUpgradeable, which are upgradeable abstract contracts, inheriting contracts may introduce new variables. In order to be able to add new variables to the upgradeable abstract contract without causing storage collisions, a storage gap should be added to the upgradeable abstract contract.

If no storage gap is added, when the upgradable abstract contract introduces new variables, it may override the variables in the inheriting contract.

Proof of Concept

https://github.com/prepo-io/prepo-monorepo/blob/9ed5587e557a1189df6c5bb0229ad8c171eaabce/packages/prepo-shared-contracts/contracts/SafeAccessControlEnumerableUpgradeable.sol#L7-L9

https://github.com/prepo-io/prepo-monorepo/blob/9ed5587e557a1189df6c5bb0229ad8c171eaabce/packages/prep

o-shared-contracts/contracts/SafeOwnableUpgradeable.sol#L7-L8

Recommended Mitigation Steps

Consider adding a storage gap at the end of the upgradeable abstract contract

```
uint256[50] private __gap;
```

prePO (sponsor) confirmed and resolved:

> Fixed in PR 356.

cccz (warden) reviewed mitigation:

> Fixed by adding uint256[50] `private __gap` in abstract upgradeable smart contracts to ensure no storage shifting down storage in the inheritance chain of the inheriting contracts.

Informational Findings (3)

[Info-01] TokenShop: _contractToIdToPrice should be reset after user purchases NFT

In the purchase function of a TokenShop contract, `_contractToIdToPrice` is not reset after the NFT is sold. In some cases this may cause the contract to suffer a loss.

Consider the following scenario:

- The TokenShop contract sells an NFT for 1000 USDC.
- After some time, the price of the NFT rises to 3000 USDC.
- The owner of the TokenShop contract buys the NFT again from the market and sends it to the contract to be sold.
- Since the current _contractToIdToPrice of the NFT is still 1000 USDC, the user can buy the NFT at 1000 USDC before the owner sets a new price.

Proof of Concept

https://github.com/prepo-io/prepo-monorepo/blob/9ed5587e557a1189df6c5bb0229ad8c171eaabce/apps/smart-contracts/token/contracts/token-shop/TokenShop.sol#L35-L53

Recommended Mitigation Steps

For ERC721, reset `_contractToIdToPrice` when the id is sold.

For ERC1155, reset `_contractToIdToPrice` when the id has a balance of 0 in the contract

[Info-02] WithdrawERC721: Using the transferFrom function of an ERC721 contract may freeze the user's NFT

When using the `transferFrom` function of an ERC721 contract to send an NFT, if the receiving address is a smart contract and does not support ERC721, the NFT can be frozen in the contract.

Proof of Concept

https://github.com/prepo-io/prepo-monorepo/blob/9ed5587e557a1189df6c5bb0229ad8c171eaabce/packages/prepo-shared-contracts/contracts/WithdrawERC721.sol#L14-L15

Recommended Mitigation Steps

Use the ERC721 contract's `safeTransferFrom` function to send NFTs.

[Info-03] Centralization problems in Vesting contract

In the Vesting contract, there are some centralization problems:

```
a) The owner can change _token at any time
b) The owner can reduce the recipient's vested amount at
c) The owner can withdraw the vested tokens in the contr
d) The owner can extend the vesting end time indefinitel
```

Since these problems can cause the recipient to suffer losses, the recipient may not trust the Vesting Contract.

Proof of Concept

https://github.com/prepo-io/prepo-monorepo/blob/9ed5587e557a1189df6c5bb0229ad8c171eaabce/apps/smart-contracts/token/contracts/vesting/Vesting.sol#L23-L59

https://github.com/prepo-io/prepo-monorepo/blob/9ed5587e557a1189df6c5bb0229ad8c171eaabce/packages/prepo-shared-contracts/contracts/WithdrawERC20.sol#L14-L19

Recommended Mitigation Steps

Consider limiting the owner's abilities.

Gas Optimizations [13]

[G-01] MiniSales.purchase(): SHOULD USE MEMORY INSTEAD OF STORAGE VARIABLE

See @audit tag

```
function purchase(address _recipient, uint256 _saleTokenAmount
  require(_purchasePrice == _price, "Price mismatch");
  if (address(_purchaseHook) != address(0)) { _purchaseHook.hc
  uint256 _paymentTokenAmount = (_saleTokenAmount * _price) /
  _paymentToken.transferFrom(_msgSender(), address(this), _pay
  _saleToken.transfer(_recipient, _saleTokenAmount);
  emit Purchase(_msgSender(), _recipient, _saleTokenAmount, _p
}
```

prePO (sponsor) confirmed and resolved:

> Fixed in PR 357.

cccz (warden) reviewed mitigation:

> Fixed by using `_purchasePrice` (memory variable) instead of `_price` (storage variable).

[G-02] MiniSales.purchase(): _purchaseHook SHOULD GET CACHED

See @audit tag

```
function purchase(address _recipient, uint256 _saleTokenAmount
  require(_purchasePrice == _price, "Price mismatch");
  if (address(_purchaseHook) != address(0)) // //@audit gas: s
      { _purchaseHook.hook(_msgSender(), _recipient, _saleToke
  uint256 _paymentTokenAmount = (_saleTokenAmount * _price) /
  _paymentToken.transferFrom(_msgSender(), address(this), _pay
  _saleToken.transfer(_recipient, _saleTokenAmount);
  emit Purchase(_msgSender(), _recipient, _saleTokenAmount, _p
}
```

prePO (sponsor) confirmed and resolved:

> Fixed in PR 357.

cccz (warden) reviewed mitigation:

> Fixed along with [G-01] in PR#357 by caching `_purchaseHook` (storage variable loaded twice).

[G-03] TokenShop.purchase(): _purchaseHook SHOULD GET CACHED

See @audit tag

```
function purchase(address[] memory _tokenContracts, uint256[]
  require(_tokenContracts.length == _amounts.length && _ids.le
  require(address(_purchaseHook) != address(0), "Purchase hook
  for (uint256 i; i < _tokenContracts.length; ++i) {
```

```
      require(_contractToIdToPrice[_tokenContracts[i]][_ids[i]]
      uint256 _totalPaymentAmount = _contractToIdToPrice[_tokenC
      _paymentToken.transferFrom(_msgSender(), address(this), _t
      bool _isERC1155 = IERC1155(_tokenContracts[i]).supportsInt
      if (_isERC1155) {
        _purchaseHook.hookERC1155(msg.sender, _tokenContracts[i]
        _userToERC1155ToIdToPurchaseCount[msg.sender][_tokenCont
        IERC1155(_tokenContracts[i]).safeTransferFrom(address(th
      } else {
        _purchaseHook.hookERC721(msg.sender, _tokenContracts[i],
        ++_userToERC721ToPurchaseCount[msg.sender][_tokenContrac
        IERC721(_tokenContracts[i]).safeTransferFrom(address(thi
      }
    }
  }
```

prePO (sponsor) confirmed and resolved:

> Fixed in PR 358.

cccz (warden) reviewed mitigation:

> Fixed by caching `_purchaseHook` (storage variable used in Loop).

[G-04] TokenShop.purchase(): _contractToIdToPrice SHOULD GET CACHED

See @audit tag

```
  function purchase(address[] memory _tokenContracts, uint256[]
    require(_tokenContracts.length == _amounts.length && _ids.le
    require(address(_purchaseHook) != address(0), "Purchase hook
    for (uint256 i; i < _tokenContracts.length; ++i) {
      require(_contractToIdToPrice[_tokenContracts[i]][_ids[i]]
      uint256 _totalPaymentAmount = _contractToIdToPrice[_tokenC
```

cccz (warden) reviewed mitigation:

> Fixed in PR#355 by caching `_contractToIdToPrice[tokenContracts[i]][ids[i]]` to local variable `_price`.

[G-05] BlocklistTransferHook.hook(): _blocklist SHOULD GET CACHED

See @audit tag

```
function hook(address _from, address _to, uint256 _amount) pub
  require(!_blocklist.isIncluded(_from), "Sender blocked"); //
  require(!_blocklist.isIncluded(_to), "Recipient blocked"); /
}
```

prePO (sponsor) confirmed and resolved:

> Fixed in PR 359.

cccz (warden) reviewed mitigation:

> Fixed by caching `_blocklist` (storage variable loaded twice).

[G-06] PurchaseHook.hookERC721(): _tokenShop SHOULD GET CACHED

See @audit tag

```
function hookERC721(address _user, address _tokenContract, uir
  require(address(_tokenShop) != address(0), "Token shop not s
  uint256 _maxPurchaseAmount = _erc721ToMaxPurchasesPerUser[_t
  if (_maxPurchaseAmount != 0) {
    require(
      _tokenShop.getERC721PurchaseCount(_user, _tokenContract)
        _maxPurchaseAmount,
      "ERC721 purchase limit reached"
    );
  }
}
```

prePO (sponsor) confirmed and resolved:

> Fixed in PR 360.

cccz (warden) reviewed mitigation:

> Fixed by caching `_tokenShop` (storage variable loaded twice) in `hookERC1155()`.

[G-07] PurchaseHook.hookERC1155(): _tokenShop SHOULD GET CACHED

See @audit tag

```
function hookERC1155(address _user, address _tokenContract, ui
  require(address(_tokenShop) != address(0), "Token shop not s
  uint256 _maxPurchaseAmount = _erc1155ToIdToMaxPurchasesPerUs
    _tokenContract
  ][_tokenId];
  if (_maxPurchaseAmount != 0) {
    require(
      _tokenShop.getERC1155PurchaseCount(_user, _tokenContract
        _amount <=
        _maxPurchaseAmount,
      "ERC1155 purchase limit reached"
    );
  }
}
```

prePO (sponsor) confirmed and resolved:

> Fixed in PR 360.

cccz (warden) reviewed mitigation:

> Fixed by caching `_tokenShop` (storage variable loaded twice) in `hookERC721()`.

[G-08] Vesting.claim(): _token SHOULD GET CACHED

See @audit tag

```
function claim() external override nonReentrant whenNotPaused
  uint256 _claimableAmount = getClaimableAmount(msg.sender);
  require(_claimableAmount > 0, "Claimable amount = 0");
  require(_token.balanceOf(address(this)) >= _claimableAmount,
  _recipientToClaimedAmount[msg.sender] += _claimableAmount;
  _token.transfer(msg.sender, _claimableAmount); //@audit gas:
  emit Claim(msg.sender, _claimableAmount);
}
```

prePO (sponsor) confirmed and resolved:

> Fixed in PR 361.

cccz (warden) reviewed mitigation:

> Fixed by caching _token (storage variable loaded twice) in claim().

[G-09] WithdrawERC20.WithdrawERC20 : owner() SHOULD GET CACHED

See @audit tag

```
function withdrawERC20(address[] calldata _erc20Tokens, uint25
  require(_erc20Tokens.length == _amounts.length, "Array lengt
  for (uint256 i; i < _erc20Tokens.length; ++i) {
    IERC20(_erc20Tokens[i]).safeTransfer(owner(), _amounts[i])
  }
}
```

prePO (sponsor) confirmed and resolved:

> Fixed in PR 362.

cccz (warden) reviewed mitigation:

> Fixed by caching `owner()` (storage variable loaded in loop) in `withdrawERC20()`.

[G-10] Vesting.setAllocations(): L50 AND L52 SHOULD BE UNCHECKED DUE TO L49

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an unchecked block: https://docs.soliditylang.org/en/v0.8.7/control-structures.html#checked-or-unchecked-arithmetic

```
    function setAllocations(address[] calldata _recipients, uint25
      require(_recipients.length == _amounts.length, "Array length
      uint256 _newTotalAllocatedSupply = _totalAllocatedSupply;
      for (uint256 i; i < _recipients.length; ++i) {
        uint256 _amount = _amounts[i];
        address _recipient = _recipients[i];
        uint256 _prevAllocatedAmount = _recipientToAllocatedAmount
        /**
         * If the new allocation amount is greater than _prevAlloc
         * the absolute difference is added to
         * _newTotalAllocatedSupply, otherwise it is subtracted.
         */
49:      if (_amount > _prevAllocatedAmount) {
50:        _newTotalAllocatedSupply += _amount - _prevAllocatedA
51:      } else {
52:        _newTotalAllocatedSupply -= _prevAllocatedAmount - _a
      }
```

prePO (sponsor) confirmed and resolved:

> Fixed in PR 361.

cccz (warden) reviewed mitigation:

> Fixed along with [G-08] in PR#361 by using unchecked for calculation of `_newTotalAllocatedSupply` in `setAllocations()`.

[G-11] INCREMENTS CAN BE UNCHECKED

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

https://github.com/ethereum/solidity/issues/10695

Instances include:

```
AccountList.set():                for (uint256 i; i < _accounts.le
AccountList.reset():     for (uint256 i; i < _newIncludedAccounts
PurchaseHook.setMaxERC721PurchasesPerUser():     for (uint256 i;
PurchaseHook.setMaxERC1155PurchasesPerUser():    for (uint256 i;
TokenShop.setContractToIdToPrice():      for (uint256 i; i < _tok

TokenShop.purchase():    for (uint256 i; i < _tokenContracts.leng
Vesting.setAllocations():         for (uint256 i; i < _recipients.
WithdrawERC20.WithdrawERC20():   for (uint256 i; i < _erc20Tokens
withdrawERC721.withdrawERC721():         for (uint256 i; i < _erc
withdrawERC1155.withdrawERC1155():       for (uint256 i; i < _erc
```

The code would go from:

```
for (uint256 i; i < numIterations; ++i) {
 // ...
}
```

to

```
for (uint256 i; i < numIterations;) {
 // ...
 unchecked { ++i; }
}
```

prePO (sponsor) confirmed and resolved:

> Fixed in PR 365.

cccz (warden) reviewed mitigation:

> Fixed by using unchecked{ ++i;} in loop to save around 80 gas per iteration.

[G-12] Vesting.claim(): > 0 IS LESS EFFICIENT THAN != 0 FOR UNSIGNED INTEGERS

!= 0 costs less gas compared to > 0 for unsigned integers in require statements with the optimizer enabled (6 gas)

```
function claim() external override nonReentrant whenNotPaused
  uint256 _claimableAmount = getClaimableAmount(msg.sender);
  require(_claimableAmount > 0, "Claimable amount = 0"); // @a
```

prePO (sponsor) confirmed and resolved:

> Fixed in PR 361.

cccz (warden) reviewed mitigation:

> Fixed along with [G-08] in PR#361 by using `!= 0` instead of `>0` in `claim()`.

[G-13] AN ARRAY'S LENGTH SHOULD BE CACHED TO SAVE GAS IN FOR-LOOPS

Reading array length at each iteration of the loop takes 6 gas (3 for mload and 3 to place memory_offset) in the stack.

Caching the array length in the stack saves around 3 gas per iteration.

Here, I suggest storing the array's length in a variable before the for-loop, and use it instead:

```
AccountList.set():              for (uint256 i; i < _accounts.le
AccountList.reset():    for (uint256 i; i < _newIncludedAccounts
PurchaseHook.setMaxERC721PurchasesPerUser():    for (uint256 i;
PurchaseHook.setMaxERC1155PurchasesPerUser():   for (uint256 i;
```

```
TokenShop.setContractToIdToPrice():     for (uint256 i; i < _tok

TokenShop.purchase():   for (uint256 i; i < _tokenContracts.leng
Vesting.setAllocations():       for (uint256 i; i < _recipients.
WithdrawERC20.WithdrawERC20():  for (uint256 i; i < _erc20Tokens
withdrawERC721.withdrawERC721():        for (uint256 i; i < _erc
withdrawERC1155.withdrawERC1155():      for (uint256 i; i < _erc
```

prePO (sponsor) confirmed and resolved:

> Fixed in PR 363.

cccz (warden) reviewed mitigation:

> Fixed by caching the array length in the stack to save around 3 gas per iteration.

Mitigation Review

Mitigation review by cccz

Mitigation Overview

The following is a high-level overview of the core changes introduced as the mitigation, arranged per the report findings.

- [M-01] Acknowledged. Only security-compliant tokens such as USDC and PPO are currently planned to be used in the contracts.
- [M-02] Fixed in PR#355. Adding purchasePrices parameter in purchase() to avoid race condition and ensure that tokens are not purchased at higher prices than the user intended.
- [M-03] Fixed in PR#352. Removing all write methods from SafeOwnableCaller and SafeAccessControlEnumerableCaller to ensure that the inheriting contracts need to override all write methods. This will avoid missing out of access control on some critical write methods.
- [M-04] Fixed in PR#356. Adding uint256[50] private __gap in abstract upgradeable smart contracts to ensure no storage shifting down storage in the inheritance chain of the inheriting contracts.
- [Info-01] Acknowledged.

- [Info-02] Acknowledged.
- [Info-03] Acknowledged.
- [G-01] Fixed in PR#357. Using _purchasePrice (memory variable) instead of _price (storage variable)
- [G-02] Fixed along with [G-01] in PR#357. Caching _purchaseHook (storage variable loaded twice)
- [G-03] Fixed in PR#358. Caching _purchaseHook (storage variable used in Loop)
- [G-04] Fixed in PR#355. Cached *contractToldToPrice[*tokenContracts[i]][_ids[i]] to local variable _price.
- [G-05] Fixed in PR#359. Caching _blocklist (storage variable loaded twice)
- [G-06] Fixed in PR#360. Caching _tokenShop (storage variable loaded twice) in hookERC1155()
- [G-07] Fixed along with [G-06] in PR#360. Caching _tokenShop (storage variable loaded twice) in hookERC721().
- [G-08] Fixed in PR#361. Caching _token (storage variable loaded twice) in claim()
- [G-09] Fixed in PR#352. Caching owner() (storage variable loaded in loop) in withdrawERC20()
- [G-10] Fixed along with [G-08] in PR#361. Using unchecked for calculation of _newTotalAllocatedSupply in setAllocations().
- [G-11] Fixed in PR#365. Using unchecked{ ++i;} in loop to save around 80 gas per iteration.
- [G-12] Fixed along with [G-08] in PR#361. Using != 0 instead of >0 in claim()
- [G-13] Fixed in PR#363. Caching the array length in the stack to save around 3 gas per iteration.

Disclosures

C4 is an open organization governed by participants in the community.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility

of users.

Top