Ondo Finance Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditor

Dacian

April 18, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Ondo Finance is a Real-World Asset Tokenization protocol which aims to bring institutional-grade investment products onto blockchains.

This audit only concerned:

- `OUSG` a tokenized wrapper of the Blackrock short-term US Treasuries ETF
- `rOUSG` a rebalancing wrapper for `OUSG`
- `OUSGInstantManager` a manager contract that allows users to mint and redeem `OUSG` and `rOUSG`
- a couple of rate-limiting contracts

General Overview:

Users who have satisfied KYC and other applicable regulatory requirements are able to use `OUSGInstantManager` to:

- mint `OUSG`/`rOUSG` in exchange for `USDC`
- redeem `OUSG`/`rOUSG` to receive `USDC`

`OUSG` and `rOUSG` function as `ERC20` tokens which users can transfer only to other KYC'd users and the protocol admin can place time-based global/individual user limits on the amount of allowed mints and redemptions.

The `USD` price of `OUSG` is set every day by an off-chain Oracle controlled by the protocol and the price is expected to be "up only" to reflect the yield generated by the underlying real-world assets.

External Integrations:

The protocol's primary external integration is with Blackrock's new `BUIDL` token and associated redemption contract. When users redeem `OUSG`/`rOUSG` via `OUSGInstantManager` the protocol must provide them with `USDC`. In order to source this `USDC` the protocol redeems enough `BUIDL` from Blackrock's redemption contract to cover the required `USDC` as Blackrock's redemption contract provides 1:1 `BUIDL:USDC` redemptions.

There is no on-chain mechanism for the protocol to deposit BUIDL with Blackrock; this process happens off-chain and we are to assume there will always be sufficient BUIDL to redeem such that the protocol will always be able to provide USDC when users redeem OUSG/rOUSG.

Centralization Risks:

Due to the regulated nature of the underlying assets being tokenized and applicable regulatory requirements, the protocol is highly centralized by design including the ability for the protocol admins to seize the assets of users; users must place a high degree of trust in the protocol team. All issues related to centralization were outside the scope of the audit.

5 Audit Scope

The following contracts were included in the scope for this audit:

```
contracts/ousg/ousgInstantManager.sol
contracts/ousg/rOUSG.sol
contracts/ousg/InvestorBasedRateLimiter.sol
contracts/InstantMintTimeBasedRateLimiter.sol
contracts/kyc/KYCRegistryClientUpgradeable.sol
```

6 Executive Summary

Over the course of 9 days, the Cyfrin team conducted an audit on the Ondo Finance smart contracts provided by Ondo Finance. In this period, a total of 24 issues were found.

The protocol was recently audited in an audit contest and we were auditing a version of the codebase which had resolved the most important findings from that contest with knowledge of the contest findings. There was 1 new contract InvestorBasedRateLimiter.sol which was not present in the audit contest.

Our findings consisted of 7 Low severity issues with the remainder being informational and gas optimizations. Of the 7 Low severity issues:

- 2 related to unlikely scenarios that could arise in the integration with Blackrock's BUIDL redemption contract
- 2 related to breaking a InvestorBasedRateLimiter invariant: "when a new investorId is created, it should be associated with one or more valid addresses"
- 1 related to breaking a ROUSG invariant: "when unwrapping users should receive the same amount of OUSG input tokens they provided when they wrapped, irrespective of price"
- 2 related to underflow reverts inside checks performed by InvestorBasedRateLimiter and InstantMintTimeBasedRateLimiter

All of the above findings were successfully mitigated by the protocol team.

Fuzz Testing:

As part of our audit we used both stateless and stateful/invariant fuzz testing; all code for our fuzz testing was delivered to the protocol team as an additional deliverable at the conclusion of the audit.

Summary

Project Name	Ondo Finance
Repository	rwa-internal
Commit	6747ebada1c8...
Audit Timeline	Apr 8th - Apr 18th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	7
Informational	7
Gas Optimizations	10
Total Issues	24

Summary of Findings

[L-1] `InvestorBasedRateLimiter::setInvestorMintLimit` and `setInvestorRedemptionLimit` can make subsequent calls to `checkAndUpdateMintLimit` and `checkAndUpdateRedemptionLimit` revert due to underflow	Resolved
[L-2] Prevent creating an investor record associated with the zero address	Resolved
[L-3] Prevent creating an investor record associated with no address	Resolved
[L-4] `InstantMintTimeBasedRateLimiter::_setInstantMintLimit` and `_setInstantRedemptionLimit` can make subsequent calls to `_checkAndUpdateInstantMintLimit` and `_checkAndUpdateInstantRedemptionLimit` revert due to underflow	Resolved
[L-5] `OUSGInstantManager` redemptions will be bricked if BlackRock deploys a new `BUIDLRedeemer` contract and sunsets the existing one	Resolved
[L-6] `ROUSG::unwrap` can unnecessarily return slightly less `OUSG` tokens than users originally wrapped	Resolved
[L-7] Protocol may be short-changed by `BuidlRedeemer` during a USDC depeg event	Resolved
[I-1] Consider implementing unlimited approvals for `rOUSG` token	Acknowledged
[I-2] Reduce approval before transferring tokens in `rOUSG::transferFrom`	Acknowledged
[I-3] Transfer tokens before minting shares in `rOUSG::wrap`	Acknowledged
[I-4] Round up fees in `OUSGInstantManager::_getInstantMintFees` and `_getInstantRedemptionFees` to favor the protocol	Acknowledged

[I-5] Misleading events are emitted when transferring a dust amount of rOUSG shares	Acknowledged
[I-6] Consider allowing `ROUSG::burn` to burn dust amounts	Resolved
[I-7] `_assertUSDCPrice` breaks the solidity style guide	Resolved
[G-1] Cache array length outside of loops and consider unchecked loop incrementing	Acknowledged
[G-2] Cache storage variables in stack when read multiple times without being changed	Acknowledged
[G-3] Avoid unnecessary initialization to zero	Resolved
[G-4] `InvestorBasedRateLimiter::_initializeInvestorState` should return newly created `investorId` to save re-reading it from storage	Resolved
[G-5] Refactor `InvestorBasedRateLimiter::checkAndUpdateMintLimit` and `checkAndUpdateRedemptionLimit` to avoid performing unnecessary operations when creating a new investor	Acknowledged
[G-6] In `InvestorBasedRateLimiter::_setAddressToInvestorId` first read `addressToInvestorId[investorAddress]` then use it in the `if` statement check	Acknowledged
[G-7] In `InvestorBasedRateLimiter::_setAddressToInvestorId` use `delete` when setting to zero for gas refund	Acknowledged
[G-8] Remove return parameters from `rOUSG::_mintShares` and `_burnShares` as they are never read	Resolved
[G-9] In `OUSGInstantManager::_mint` and `_redeem` cache `feeReceiver` and only emit fee event if fees are deducted	Acknowledged
[G-10] Change `ROUSG::unwrap` to return amount of `OUSG` output tokens then use that as input when calling `_redeem` in `OUSGInstantManager::redeemRebasingOUSG`	Acknowledged

7 Findings

7.1 Low Risk

7.1.1 `InvestorBasedRateLimiter::setInvestorMintLimit` and `setInvestorRedemptionLimit` can make subsequent calls to `checkAndUpdateMintLimit` and `checkAndUpdateRedemptionLimit` revert due to underflow

Description: `InvestorBasedRateLimiter::_checkAndUpdateRateLimitState` L211-213 subtracts the current mint/redemption amount from the corresponding limit:

```
if (amount > rateLimit.limit - rateLimit.currentAmount) {
  revert RateLimitExceeded();
}
```

If `setInvestorMintLimit` or `setInvestorRedemptionLimit` are used to set the limit amount for minting or redemptions smaller than the current mint/redemption amount, calls to this function will revert due to underflow.

Impact: `InvestorBasedRateLimiter::setInvestorMintLimit` and `setInvestorRedemptionLimit` can make subsequent calls to `checkAndUpdateMintLimit` and `checkAndUpdateRedemptionLimit` revert due to underflow.

Proof of Concept: Add this drop-in PoC to `forge-tests/ousg/InvestorBasedRateLimiter/setters.t.sol`:

```
function test_setInvestorMintLimit_underflow_DoS() public initDefault(alice) {
    // first perform a mint
    uint256 mintAmount = rateLimiter.defaultMintLimit();
    vm.prank(client);
    rateLimiter.checkAndUpdateMintLimit(alice, mintAmount);

    // admin now reduces the mint limit to be under the current
    // minted amount
    uint256 aliceInvestorId = 1;
    uint256 newMintLimit = mintAmount - 1;
    vm.prank(guardian);
    rateLimiter.setInvestorMintLimit(aliceInvestorId, newMintLimit);

    // subsequent calls to `checkAndUpdateMintLimit` revert due to underflow
    vm.prank(client);
    rateLimiter.checkAndUpdateMintLimit(alice, 1);

    // same issue affects `setInvestorRedemptionLimit`
}
```

Run with: `forge test --match-test test_setInvestorMintLimit_underflow_DoS`

Produces output:

```
Ran 1 test for
↪  forge-tests/ousg/InvestorBasedRateLimiter/setters.t.sol:Test_InvestorBasedRateLimiter_setters_ETH
[FAIL. Reason: panic: arithmetic underflow or overflow (0x11)]
↪  test_setInvestorMintLimit_underflow_DoS() (gas: 264384)
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 1.09ms (116.74µs CPU time)
```

Recommended Mitigation: Explicitly handle the case where the limit is smaller than the current mint/redemption amount:

```
if (rateLimit.limit <= rateLimit.currentAmount || amount > rateLimit.limit - rateLimit.currentAmount) {
  revert RateLimitExceeded();
```

```
}
```

Ondo: Fixed in commit fb8ecff.

Cyfrin: Verified.

7.1.2 Prevent creating an investor record associated with the zero address

Description: `InvestorBasedRateLimiter::checkAndUpdateMintLimit` and `checkAndUpdateRedemptionLimit` can create a new investor record and associate it with the zero address.

Impact: Investor records can be created which are associated with the zero address. This breaks the following invariant of the `InvestorBasedRateLimiter` contract:

> when a new `investorId` is created, it should be associated with one or more valid addresses

Proof of Concept: Add this drop-in PoC to `forge-tests/ousg/InvestorBasedRateLimiter/client.t.sol`:

```
function test_mint_zero_address() public {
    uint256 mintAmount = rateLimiter.defaultMintLimit();
    vm.prank(client);
    rateLimiter.checkAndUpdateMintLimit(address(0), mintAmount);

    // an investor has been created with a 0 address
    assertEq(1, rateLimiter.addressToInvestorId(address(0)));

    // same issue affects checkAndUpdateRedemptionLimit
}
```

Run with: `forge test --match-test test_mint_zero_address`

Recommended Mitigation: In `_setAddressToInvestorId` revert for the zero address:

```
function _setAddressToInvestorId(
    address investorAddress,
    uint256 newInvestorId
) internal {
    if(investorAddress == address(0)) revert NoZeroAddress();
```

Ondo: Fixed in commit bac99d0.

Cyfrin: Verified.

7.1.3 Prevent creating an investor record associated with no address

Description: `InvestorBasedRateLimiter::initializeInvestorStateDefault` is supposed to associate a newly created investor with one or more addresses but the `for` loop which does this can be bypassed by calling the function with an empty array:

```
function initializeInvestorStateDefault(
    address[] memory addresses
    ) external onlyRole(CONFIGURER_ROLE) {
    _initializeInvestorState(
      addresses,
      defaultMintLimit,
      defaultRedemptionLimit,
      defaultMintLimitDuration,
      defaultRedemptionLimitDuration
    );
}
```

```
function _initializeInvestorState(
    address[] memory addresses,
    uint256 mintLimit,
    uint256 redemptionLimit,
    uint256 mintLimitDuration,
    uint256 redemptionLimitDuration
    ) internal {
    uint256 investorId = ++investorIdCounter;

    // @audit this `for` loop can by bypassed by calling
    // `initializeInvestorStateDefault` with an empty array
    for (uint256 i = 0; i < addresses.length; ++i) {
      // Safety check to ensure the address is not already associated with an investor
      // before associating it with a new investor
      if (addressToInvestorId[addresses[i]] != 0) {
        revert AddressAlreadyAssociated();
      }
      _setAddressToInvestorId(addresses[i], investorId);
    }

    investorIdToMintState[investorId] = RateLimit({
      currentAmount: 0,
      limit: mintLimit,
      lastResetTime: block.timestamp,
      limitDuration: mintLimitDuration
    });
    investorIdToRedemptionState[investorId] = RateLimit({
      currentAmount: 0,
      limit: redemptionLimit,
      lastResetTime: block.timestamp,
      limitDuration: redemptionLimitDuration
    });
}
```

Impact: An investor record can be created without any associated address. This breaks the following invariant of the `InvestorBasedRateLimiter` contract:

> when a new `investorId` is created, it should be associated with one or more valid addresses

Proof of Concept: Add this drop-in PoC to `forge-tests/ousg/InvestorBasedRateLimiter/setters.t.sol`:

```
function test_initializeInvestor_NoAddress() public {
    // no investor created
    assertEq(0, rateLimiter.investorIdCounter());

    // empty input array will bypass the `for` loop that is supposed
    // to associate addresses to the newly created investor
    address[] memory addresses;

    vm.prank(guardian);
    rateLimiter.initializeInvestorStateDefault(addresses);

    // one investor created
    assertEq(1, rateLimiter.investorIdCounter());

    // not associated with any addresses
    assertEq(0, rateLimiter.investorAddressCount(1));
}
```

Run with: `forge test --match-test test_initializeInvestor_NoAddress`

Recommended Mitigation: In `_initializeInvestorState` revert if the input address array is empty:

```
uint256 addressesLength = addresses.length;

if(addressesLength == 0) revert EmptyAddressArray();
```

Ondo: Fixed in commit bac99d0.

Cyfrin: Verified.

7.1.4 `InstantMintTimeBasedRateLimiter::_setInstantMintLimit` and `_setInstantRedemptionLimit` can make subsequent calls to `_checkAndUpdateInstantMintLimit` and `_checkAndUpdateInstantRedemptionLimit` revert due to underflow

Description: `InstantMintTimeBasedRateLimiter::_checkAndUpdateInstantMintLimit` L103-106 subtracts the currently minted amount from the mint limit:

```
require(
  amount <= instantMintLimit - currentInstantMintAmount,
  "RateLimit: Mint exceeds rate limit"
);
```

If `_setInstantMintLimit` is used to set `instantMintLimit < currentInstantMintAmount`, subsequent calls to this function will revert due the underflow. The same is true for `_setInstantRedemptionLimit` and `_checkAndUpdateInstantRedemptionLimit`.

Impact: `InstantMintTimeBasedRateLimiter::_setInstantMintLimit` and `_setInstantRedemptionLimit` can make subsequent calls to `_checkAndUpdateInstantMintLimit` and `_checkAndUpdateInstantRedemptionLimit` revert due to underflow.

Recommended Mitigation: Explicitly handle the case where the limit is smaller than the current mint/redemption amount:

```
function _checkAndUpdateInstantMintLimit(uint256 amount) internal {
    require(
      instantMintLimit > currentInstantMintAmount && amount <= instantMintLimit -
      ↪  currentInstantMintAmount,
      "RateLimit: Mint exceeds rate limit"
    );
}

function _checkAndUpdateInstantRedemptionLimit(uint256 amount) internal {
    require(
      instantRedemptionLimit > currentInstantRedemptionAmount && amount <= instantRedemptionLimit -
      ↪  currentInstantRedemptionAmount,
      "RateLimit: Redemption exceeds rate limit"
    );
}
```

Ondo: Fixed in commit fb8ecff.

Cyfrin: Verified.

7.1.5 `OUSGInstantManager` redemptions will be bricked if BlackRock deploys a new `BUIDLRedeemer` contract and sunsets the existing one

Description: The `BUIDLRedeemer` contract is a very new contract; it is very possible that in the future a new version of the contract will be deployed and the current version will cease to function.

To future-proof `OUSGInstantManager` and ensure it will continue to function in this situation, remove the `immutable` keyword from the `buidlRedeemer` definition and add a setter function that allows it to be updated in the future.

Ondo: If a new `BUIDLRedeemer` contract is deployed our plan is to deploy a new `OUSGInstantManager`. We prefer to make it harder for us to change the address of `buidlRedeemer` to ensure there is proper due diligence of any changes.

7.1.6 `ROUSG::unwrap` can unnecessarily return slightly less `OUSG` tokens than users originally wrapped

Description: One invariant of the `ROUSG` token is:

> when unwrapping users should receive the same amount of OUSG input tokens they provided when they wrapped, irrespective of price

However this can often not be the case as `ROUSG::unwrap` can unnecessarily return slightly less `OUSG` tokens than users originally wrapped.

Impact: Users will unnecessarily receive slightly less tokens than they originally wrapped, breaking an invariant of the `ROUSG` contract.

Proof of Concept: Run this stand-alone stateless fuzz test which shows the problem:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.23;

import "forge-std/Test.sol";

// run from base project directory with:
// forge test --match-contract ROUSGWrapUnwrapBrokenInvariantTest -vvv

contract ROUSGWrapUnwrapBrokenInvariantTest is Test {

    uint256 public constant OUSG_TO_ROUSG_SHARES_MULTIPLIER = 10_000;

    function _getROUSGByShares(uint256 _shares, uint256 ousgPrice) internal pure returns (uint256
    ↪  rOUSGAmount) {
        rOUSGAmount = (_shares * ousgPrice) / (1e18 * OUSG_TO_ROUSG_SHARES_MULTIPLIER);
    }

    function getSharesByROUSG(uint256 _rOUSGAmount, uint256 ousgPrice)
    internal pure returns (uint256 shares) {
        shares = (_rOUSGAmount * 1e18 * OUSG_TO_ROUSG_SHARES_MULTIPLIER) / ousgPrice;
    }

    function _wrap(uint256 _OUSGAmount) internal pure returns (uint256 shares) {
        require(_OUSGAmount > 0, "rOUSG: can't wrap zero OUSG tokens");

        shares = _OUSGAmount * OUSG_TO_ROUSG_SHARES_MULTIPLIER;
    }

    function _unwrap(uint256 _rOUSGAmount, uint256 ousgPrice) internal pure returns(uint256 tokens) {
        require(_rOUSGAmount > 0, "rOUSG: can't unwrap zero rOUSG tokens");

        uint256 ousgSharesAmount = getSharesByROUSG(_rOUSGAmount, ousgPrice);

        vm.assume(ousgSharesAmount >= OUSG_TO_ROUSG_SHARES_MULTIPLIER);
```

```
        tokens = ousgSharesAmount / OUSG_TO_ROUSG_SHARES_MULTIPLIER;
    }

    function test_WrapUnwrapReturnsInputTokens(uint256 initialOUSGAmount, uint256 ousgPrice) external {
        // bound inputs
        initialOUSGAmount  = bound(initialOUSGAmount, 100000e18, type(uint128).max);
        ousgPrice          = bound(ousgPrice, 105e18, 106e18);

        // wrap OUSG into rOUSG
        uint256 rousgShares = _wrap(initialOUSGAmount);

        // get the token amount of rOUSG equivalent to the received shares
        uint256 rousgAmount = _getROUSGByShares(rousgShares, ousgPrice);

        // use the token amount to unwrap rOUSG back into OUSG
        uint256 finalOUSGAmount = _unwrap(rousgAmount, ousgPrice);

        // verify amounts match; this fails as user is slighty short-changed
        assertEq(initialOUSGAmount, finalOUSGAmount);
    }
}
```

Recommended Mitigation: When calling `ROUSG::unwrap`, `burn` and `OUSGInstantManager::redeemRebasingOUSG`, instead of passing in the `ROUSG` token amount the callers should pass in the share amount which can be retrieved via `ROUSG::sharesOf`. The output token calculation can then be performed as `shares / OUSG_TO_ROUSG_SHARES_MULTIPLIER` which will always return the correct amount of tokens.

The existing functions do not necessarily need to be removed but additional functions should be created to allow users to input the share amounts. The following function has been tested via an invariant fuzz testing suite and appears to always return the correct amount:

```
  // @audit this function allow unwrapping by shares instead of tokens
  // to prevent users being slightly short-changed such that users will
  // always receive the same input amount of OUSG tokens
  function unwrapShares(uint256 _shares) external whenNotPaused {
    uint256 ousgTokens = _shares / OUSG_TO_ROUSG_SHARES_MULTIPLIER;

    require(ousgTokens > 0, "rOUSG: no tokens to send, unwrap more shares");

    uint256 rousgBurned = getROUSGByShares(_shares);

    _burnShares(msg.sender, _shares);
    ousg.transfer(msg.sender, ousgTokens);

    emit Transfer(msg.sender, address(0), rousgBurned);
    emit TransferShares(msg.sender, address(0), _shares);
  }
```

Proof that this mitigation works, using a modified version of the PoC stateless fuzz test:

First ensure that `foundry.toml` has the fuzz setting increased for example:

```
[fuzz]
runs = 1000000
```

Then run this stand-alone stateless fuzz test which verifies the solution:

```
// SPDX-License-Identifier: MIT
```

```
pragma solidity ^0.8.23;

import "forge-std/Test.sol";

// run from base project directory with:
// forge test --match-contract ROUSGWrapUnwrapFixedInvariantTest -vvv

contract ROUSGWrapUnwrapFixedInvariantTest is Test {

    uint256 public constant OUSG_TO_ROUSG_SHARES_MULTIPLIER = 10_000;

    function _wrap(uint256 _OUSGAmount) internal pure returns (uint256 shares) {
        require(_OUSGAmount > 0, "rOUSG: can't wrap zero OUSG tokens");

        shares = _OUSGAmount * OUSG_TO_ROUSG_SHARES_MULTIPLIER;
    }

    function _unwrapShares(uint256 shares) internal pure returns(uint256 tokens) {
        tokens = shares / OUSG_TO_ROUSG_SHARES_MULTIPLIER;
    }

    function test_WrapUnwrapReturnsInputTokens(uint256 initialOUSGAmount, uint256 ousgPrice) external {
        // bound inputs
        initialOUSGAmount  = bound(initialOUSGAmount, 100000e18, type(uint128).max);
        ousgPrice          = bound(ousgPrice, 105e18, 106e18);

        // wrap OUSG into rOUSG
        uint256 rousgShares = _wrap(initialOUSGAmount);

        // use the token amount to unwrap rOUSG back into OUSG
        uint256 finalOUSGAmount = _unwrapShares(rousgShares);

        assertEq(initialOUSGAmount, finalOUSGAmount);
    }
}
```

Ondo: Fixed in commits df0e491, 2aa437a. We decided on not making any changes to `OUSGInstantManager` due to the amount of code changes necessary.

Cyfrin: Verified.

7.1.7 Protocol may be short-changed by `BuidlRedeemer` during a USDC depeg event

Description: `OUSGInstantManager::_redeemBUIDL` assumes that 1 BUIDL = 1 USDC as it enforces receiving 1 USDC for every 1 BUIDL it redeems:

```
uint256 usdcBalanceBefore = usdc.balanceOf(address(this));
buidl.approve(address(buidlRedeemer), buidlAmountToRedeem);
buidlRedeemer.redeem(buidlAmountToRedeem);
require(
  usdc.balanceOf(address(this)) == usdcBalanceBefore + buidlAmountToRedeem,
  "OUSGInstantManager::_redeemBUIDL: BUIDL:USDC not 1:1"
);
```

In the event of a USDC depeg (especially if the depeg is sustained), `BUIDLRedeemer` should return greater than a 1:1 ratio since 1 USDC would not be worth $1, hence 1 BUIDL != 1 USDC meaning the value of the protocol's BUIDL is worth more USDC. However `BUIDLReceiver` does not do this, it only ever returns 1:1.

Impact: In the event of a USDC depeg the protocol will be short-changed by `BuidlRedeemer` since it will happily

receive only 1 USDC for every 1 BUIDL redeemed, even though the value of 1 BUIDL would be greater than the value of 1 USDC due to the USDC depeg.

Recommended Mitigation: To prevent this situation the protocol would need to use an oracle to check whether USDC had depegged and if so, calculate the amount of USDC it should receive in exchange for its BUIDL. If it is short-changed it would either have to revert preventing redemptions or allow the redemption while saving the short-changed amount to storage then implement an off-chain process with BlackRock to receive the short-changed amount.

Alternatively the protocol may simply accept this as a risk to the protocol that it will be willingly short-changed during a USDC depeg in order to allow redemptions to continue.

Ondo: Fixed in commits 408bff1, 8a9cae9. We now use Chainlink USDC/USD Oracle and if USDC depegs below our tolerated minimum value both minting and redemptions will be stopped.

Cyfrin: Verified.

7.2 Informational

7.2.1 Consider implementing unlimited approvals for `rOUSG` token

Description: ERC20 tokens commonly implement unlimited approvals by allowing users to approve spenders for `type(uint256).max`. Consider implementing this common feature; an example from OpenZeppelin.

Ondo: Acknowledged.

7.2.2 Reduce approval before transferring tokens in `rOUSG::transferFrom`

Description: `rOUSG::transferFrom` L286-289 currently checks approvals, transfers the tokens then reduces the approvals:

```
// verify approval
require(currentAllowance >= _amount, "TRANSFER_AMOUNT_EXCEEDS_ALLOWANCE");

// perform transfer
_transfer(_sender, _recipient, _amount);

// reduce approval
_approve(_sender, msg.sender, currentAllowance - _amount);
```

A safer coding pattern is to reduce the approval first then transfer tokens similar to OpenZeppelin's impementation.

Ondo: Acknowledged.

7.2.3 Transfer tokens before minting shares in `rOUSG::wrap`

Description: `rOUSG::wrap` L411-413 currently mints shares before transferring tokens used to mint those shares:

```
// mint shares
uint256 ousgSharesAmount = _OUSGAmount * OUSG_TO_ROUSG_SHARES_MULTIPLIER;
_mintShares(msg.sender, ousgSharesAmount);

// transfer tokens used to mint the shares
ousg.transferFrom(msg.sender, address(this), _OUSGAmount);
```

A safer coding pattern is to transfer the tokens first then mint the shares.

Ondo: Acknowledged.

7.2.4 Round up fees in `OUSGInstantManager::_getInstantMintFees` and `_getInstantRedemptionFees` to favor the protocol

Description: Solidity rounds down by default so consider explicitly rounding up fees in `OUSGInstantManager::_getInstantMintFees` and `_getInstantRedemptionFees` to favor the protocol.

Ondo: Acknowledged.

7.2.5 Misleading events are emitted when transferring a dust amount of rOUSG shares

Description: Calling `ROUSG.transferShares` emits two events:

`TransferShares`: How much rOUSG shares were transferred `Transfer`: How much rOUSG tokens were transferred

Calling this function with a dust amount will emit an event that a non-zero amount of shares was transferred, together with an event that zero tokens were transferred as the `getROUSGByShares` will round to 0.

Ondo: Acknowledged.

7.2.6 Consider allowing `ROUSG::burn` to burn dust amounts

Description: `ROUSG::burn` is used by admins to burn `rOUSG` tokens from any account for regulatory reasons.

It does not allow burning a share amount smaller than 1e4, because this is less than a wei of `OUSG`.

```
if (ousgSharesAmount < OUSG_TO_ROUSG_SHARES_MULTIPLIER)
      revert UnwrapTooSmall();
```

Depending on the current and future regulatory situation it could be necessary to always be able to burn all shares from users.

Recommended Mitigation: Consider allowing the `burn` function to burn all remaining shares even if under the minimum amount.

Ondo: Fixed in commit 2aa437a.

Cyfrin: Verified.

7.2.7 `_assertUSDCPrice` breaks the solidity style guide

Description: The `_assertUSDCPrice` function is public and starts with an underscore. According to the solidity style guide, this convention is suggested for non-external functions and state variables (private or internal).

Recommended Mitigation: Remove the _, or change the visibility of the function.

Ondo: Fixed in commit fc1c8fb.

Cyfrin: Verified.

7.3 Gas Optimization

7.3.1 Cache array length outside of loops and consider unchecked loop incrementing

Description: Cache array length outside of loops and consider using `unchecked {++i;}` if not compiling with `solc --ir-optimized --optimize`:

```
File: contracts/ousg/InvestorBasedRateLimiter.sol

253:     for (uint256 i = 0; i < addresses.length; ++i) {
```

```
File: contracts/ousg/ousgInstantManager.sol

881:     for (uint256 i = 0; i < exCallData.length; ++i) {
```

Ondo: Acknowledged.

7.3.2 Cache storage variables in stack when read multiple times without being changed

Description: Reading from storage is considerably more expensive than reading from the stack so cache storage variables when read multiple times without being changed:

```
File: contracts/ousg/InvestorBasedRateLimiter.sol

// @audit cache these then use cache values when emitting event to save 2 storage reads
324:       --investorAddressCount[previousInvestorId];
335:       ++investorAddressCount[newInvestorId];

// @audit cache and use cached value for check in L470 to save 1 storage read
462:     if (mintState.lastResetTime == 0) {

// @audit cache and use cached value for check in L506 to save 1 storage read
498:     if (redemptionState.lastResetTime == 0) {
```

Ondo: Acknowledged.

7.3.3 Avoid unnecessary initialization to zero

Description: Avoid unnecessary initialization to zero:

```
File: contracts/ousg/InvestorBasedRateLimiter.sol

253:     for (uint256 i = 0; i < addresses.length; ++i) {
```

```
File: contracts/ousg/ousgInstantManager.sol

106:   uint256 public mintFee = 0;

109:   uint256 public redeemFee = 0;

881:     for (uint256 i = 0; i < exCallData.length; ++i) {
```

Ondo: Fixed in commit a7dab64.

Cyfrin: Verified.

7.3.4 `InvestorBasedRateLimiter::_initializeInvestorState` should return newly created `investorId` to save re-reading it from storage

Description: `InvestorBasedRateLimiter::_initializeInvestorState` should return the newly created `investorId`; this can then be used inside `checkAndUpdateMintLimit` and `checkAndUpdateRedemptionLimit` to save 1 storage read in each function. For example take `checkAndUpdateMintLimit`:

```
      _initializeInvestorState(
        addresses,
        defaultMintLimit,
        defaultRedemptionLimit,
        defaultMintLimitDuration,
        defaultRedemptionLimitDuration
      );

      // @audit GAS - save 1 storage read by having _initializeInvestorState
      // return the new `investorId`
      investorId = addressToInvestorId[investorAddress];
```

This can simply become:

```
investorId = _initializeInvestorState(
        addresses,
        defaultMintLimit,
        defaultRedemptionLimit,
        defaultMintLimitDuration,
        defaultRedemptionLimitDuration
      );
```

Ondo: Fixed in commit 192c7ca.

Cyfrin: Verified.

7.3.5 Refactor `InvestorBasedRateLimiter::checkAndUpdateMintLimit` and `checkAndUpdateRedemptionLimit` to avoid performing unnecessary operations when creating a new investor

Description: When creating a new investor inside `InvestorBasedRateLimiter::checkAndUpdateMintLimit` and `checkAndUpdateRedemptionLimit` there is no need to do a lot of the current processing that occurs after the second `if` statement. A more optimized version could look like this:

```
  function checkAndUpdateMintLimitOptimized(
    address investorAddress,
    uint256 mintAmount
  ) external override onlyRole(CLIENT_ROLE) {
    if (mintAmount == 0) {
      revert InvalidAmount();
    }

    uint256 investorId = addressToInvestorId[investorAddress];

    if (investorId == 0) {
      // @audit GAS - for new investor, revert if `mintAmount > defaultMintLimit`
      // otherwise execute next code then update investorIdToMintState[investorId].currentAmount
      // and slightly change emitted event since prevAmount = 0
      uint256 defaultMintLimitCache = defaultMintLimit;

      if(mintAmount > defaultMintLimitCache) revert RateLimitExceeded();

      // If this is a new investor, initialize their state with the default values
```

```
    address[] memory addresses = new address[](1);
    addresses[0] = investorAddress;

    // @audit GAS - return new investorId from `_initializeInvestorState`
    investorId = _initializeInvestorState(
      addresses,
      defaultMintLimit,
      defaultRedemptionLimit,
      defaultMintLimitDuration,
      defaultRedemptionLimitDuration
    );

    // @audit now update current minted amount
    investorIdToMintState[investorId].currentAmount = mintAmount;

    // @audit and alter emitted event to reflect first mint for this new investor
    emit MintStateUpdated(
      investorAddress,
      investorId,
      0,
      mintAmount,
      defaultMintLimitCache - mintAmount
    );
  }
  else {
    // @audit GAS - wrap remaining code in an `else` to only
    // execute if it wasn't a new investor
    RateLimit storage mintState = investorIdToMintState[investorId];

    uint256 prevAmount = mintState.currentAmount;
    _checkAndUpdateRateLimitState(mintState, mintAmount);

    emit MintStateUpdated(
      investorAddress,
      investorId,
      prevAmount,
      mintState.currentAmount,
      mintState.limit - mintState.currentAmount
    );
  }
}
```

The same optimization could be applied to `checkAndUpdateRedemptionLimit`.

Ondo: Acknowledged.

7.3.6 In `InvestorBasedRateLimiter::_setAddressToInvestorId` first read `addressToInvestorId[investorAddress]` then use it in the `if` statement check

Description: In `InvestorBasedRateLimiter::_setAddressToInvestorId` first read `addressToInvestorId[investorAddress]` then use it in the `if` statement check to save 1 storage read:

```
function _setAddressToInvestorId(
  address investorAddress,
  uint256 newInvestorId
) internal {
  // @audit GAS - do this first then use it in `if` check to save 1 storage read
  uint256 previousInvestorId = addressToInvestorId[investorAddress];

  // prevents creating the same existing association
```

```
    if (previousInvestorId == newInvestorId) {
      revert AddressAlreadyAssociated();
    }
```

Ondo: Acknowledged.

7.3.7 In `InvestorBasedRateLimiter::_setAddressToInvestorId` use `delete` when setting to zero for gas refund

Description: In `InvestorBasedRateLimiter::_setAddressToInvestorId` use `delete` when setting to zero:

```
    // If the address is not being disassociated from all investors, increment the count
    // for the investor the address is being associated with.
    if (newInvestorId != 0) {
      ++investorAddressCount[newInvestorId];

      emit AddressToInvestorIdSet(
        investorAddress,
        newInvestorId,
        investorAddressCount[newInvestorId]
      );

       // @audit move this here when setting a valid value
       addressToInvestorId[investorAddress] = newInvestorId;
    }
    else {
       // @audit use `delete` when setting to 0 for gas refund
       delete addressToInvestorId[investorAddress];
    }
```

Ondo: Acknowledged.

7.3.8 Remove return parameters from `rOUSG::_mintShares` and `_burnShares` as they are never read

Description: Remove return parameters from `rOUSG::_mintShares` and `_burnShares` as they are never read. This saves 1 storage read in each function plus the cost of the return parameters.

Ondo: Fixed in commit dc91728.

Cyfrin: Verified.

7.3.9 In `OUSGInstantManager::_mint` and `_redeem` cache `feeReceiver` and only emit fee event if fees are deducted

Description: In `OUSGInstantManager::_mint` cache `feeReceiver` and only emit fee event if fees are deducted to save 1 storage read:

```
    // Transfer USDC
    if (usdcFees > 0) {
      // @audit GAS - cache `feeReceiver` and only emit fee event if
      // fees are deducted
      address feeReceiverCached = feeReceiver;

      usdc.transferFrom(msg.sender, feeReceiverCached, usdcFees);
      emit MintFeesDeducted(msg.sender, feeReceiverCached, usdcFees, usdcAmountIn);
    }
```

A similar optimization can be made in `_redeem`.

Ondo: Acknowledged.

7.3.10 Change `ROUSG::unwrap` to return amount of `OUSG` output tokens then use that as input when calling `_redeem` in `OUSGInstantManager::redeemRebasingOUSG`

Description: Change `ROUSG::unwrap` to return amount of `OUSG` output tokens then use that as input when calling `_redeem` in `OUSGInstantManager::redeemRebasingOUSG`:

```
uint256 ousgAmountIn = rousg.unwrap(rousgAmountIn);

usdcAmountOut = _redeem(ousgAmountIn);
```

Ondo: Acknowledged.