code4rena

Liquid Ron

Smart Contract Security Assessment

Audit dates: Jan 28 — Feb 04, 2025

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 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Liquid Ron smart contract system. The audit took place from January 28 to February 04, 2025.

Following the C4 audit, 3 wardens (Aamir, ilchovski, and Orpse) reviewed the mitigations implemented by the Liquid Ron team; the mitigation review report is appended below the audit report.

The audit and mitigation review were judged by 0xsomeone.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded an aggregated total of 3 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 2 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 17 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Liquid Ron repository, and is composed of 6 smart contracts written in the Solidity programming language and includes 386 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on 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

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (1)

[H-01] The calculation of `totalAssets()` could be wrong if `operatorFeeAmount > 0`, this can cause potential loss for the new depositors

Submitted by 0x04bytes, also found by 056Security, Orpse (1, 2), 0xObserver (1, 2), 0x0dd, 0x23r0, 0xAlix2, 0xDemon, 0xG0P1, 0xRajkumar, 0xrex, 0xvd, Aamir, aariiif, Adotsam, air_0x, aldarion, Alekso, arman, attentioniayn, Bauchibred, bigbear1229, BlackAdam, Breeje (1, 2), ccvascocc, csanuragjain, curly, Darinrikusham, DemoreX, DoD4uFN, ElCid, emerald7017, EPSec, eta, Fitro, future2_22, gesha17, grearlake, harry_cryptodev, hrmneffdii, hyuunn, ilchovski, IIIIHunterIIII, JCN, Josh4324, jsonDoge, klau5, muncentu, nnez, oakcobalt (1, 2), PabloPerez, peanuts, phoenixV110, roccomania, rudhra, Ryonen, santiellena, serial-coder, Shahil_Hussain, silver_eth, sl1, spuriousdragon, stuart_the_minion, t0x1c (1, 2), Tadev, udogodwin, valy001, victortheoracle, wellbyt3, y4y, YouCrossTheLineAlfie, zainab_ou, and zraxx

LiquidRon.sol#L293-L295
LiquidRon.sol#L121-L126

Description

The fee accumulated by operator is stored in `operatorFeeAmount`. The amount is directly recorded based on the number of actual assets accumulated, not the portion of shares. The problem is, this fee is stored in the vault contract as WRON token balance together with the assets deposited by the users.

Because the calculation of `totalAssets()` will also depend on the WRON token balance owned by the vault contract, the fee withdrawn by the operator can decrease the total assets in circulation. It means that the users who withdraw their funds after the operator withdraw the fee will receive less assets than the users who withdraw before the fee withdrawal.

Impact

Potential assets loss for the users who withdraw funds after operator withdraw the fee.

Proof of Concept

To make things clear here, let's consider the following scenario. To make the scenario easier, just assume there is enough balance for the new user to withdraw.

1. The operator call `harvest()`. This will increase WRON balance owned by the vault and also increase `operatorFeeAmount`.
2. A new user deposit assets and receive shares. The calculation of `totalAssets()` will include the amount of operator's fee.
3. The operator withdraw the fee by calling `fetchOperatorFee()` function.
4. The new user withdraw his funds by calling `redeem()`. Now the user receives less assets because the calculation of `totalAssets()` will be based on the new WRON balance after fee withdrawal.

The detailed example:

Initial state:

```
totalBalance = 10000 // balance in all (vault, staked, rewards)
totalShares = s // just assume it is a variable `s` to make the
calculation easier
operatorFeeAmount = 0
```

Operator call `harvest()`:

The state of vault now:

```
totalBalance = 10000 // the total balance is not changed, just the form
is changed from rewards into actual WRON
totalShares = s
operatorFeeAmount = 10 // let's assume the operator get 10 units as fee
```

New user deposit 100 units:

The number of shares received by the new user:

```
userShares = (100*totalShares)/totalBalance
userShares = (100*s)/10000
userShares = (1/100)s
```

The step above will increase the `totalShares`.

The state of vault now:

```
totalBalance = 10100 // including the deposit by new user
totalShares = s + s/100
```

```
operatorFeeAmount = 10
```

Operator withdraws the fee:

The state of vault now:

```
totalBalance = 10090 // total balance is decreased by 10 as operator
withdraw the fee
totalShares = s + s/100
operatorFeeAmount = 0
```

The user withdraw his funds:

The assets received by the new user will be:

```
userAsset = (userShares*totalBalance)/totalShares
userAsset = ((s/100) * 10090)/(s + (s/100))
userAsset = ((s/100) * 10090)/((101/100)s)
userAsset = 10090/101
userAsset = 99.9
```

After withdrawal, the new user will receive 99.9 units. The new user loss `0.1` units.

POC Code

Copy the POC code below to `LiquidRonTest` contract in `test/LiquidRon.t.sol` and then run the test.

```
function test_withdraw_new_user() public {
        address user1 = address(0xf1);
        address user2 = address(0xf2);

        uint256 amount = 100000 ether;
        vm.deal(user1, amount);
        vm.deal(user2, amount);

        vm.prank(user1);
        liquidRon.deposit{value: amount}();

        uint256 delegateAmount = amount / 7;
        uint256[] memory amounts = new uint256[](5);
        for (uint256 i = 0; i < 5; i++) {
            amounts[i] = delegateAmount;
        }
        liquidRon.delegateAmount(0, amounts, consensusAddrs);

        skip(86400 * 365 + 2 + 1);
```

```
        // operator fee before harvest
        assertTrue(liquidRon.operatorFeeAmount() == 0);
        liquidRon.harvest(0, consensusAddrs);
        // operator fee after harvest
        assertTrue(liquidRon.operatorFeeAmount() > 0);

        // new user deposit
        vm.prank(user2);
        liquidRon.deposit{value: amount}();
        uint256 user2Shares = liquidRon.balanceOf(user2);
        uint256 expectedRedeemAmount =
liquidRon.previewRedeem(user2Shares);

        // fee withdrawal by operator
        liquidRon.fetchOperatorFee();
        assertTrue(liquidRon.operatorFeeAmount() == 0);

        // user2 redeem all his shares
        vm.prank(user2);
        liquidRon.redeem(user2Shares, user2, user2);

        console.log(user2.balance);
        console.log(expectedRedeemAmount);
        assertTrue(user2.balance == expectedRedeemAmount);
    }
```

Based on the POC code above, the last assertion `assertTrue(user2.balance == expectedRedeemAmount);` will fail because the amount withdrawn is not equal to the expected withdrawn.

Recommended Mitigation Steps

Change the formula that calculate `totalAssets()` to include `operatorFeeAmount` to subtract the total balance.

```
function totalAssets() public view override returns (uint256) {
-       return super.totalAssets() + getTotalStaked() +
getTotalRewards();
+       return super.totalAssets() + getTotalStaked() +
getTotalRewards() - operatorFeeAmount;
    }
```

Owl (Liquid Ron) confirmed and commented via duplicate issue S-174:

A simpler fix would be to include `operationFeeAmount` in total assets like such:

```
    function totalAssets() public view override returns (uint256) {
        return super.totalAssets() + getTotalStaked() + getTotalRewards()
- operationFeeAmount;
    }
```

Oxsomeone (judge) increased severity to High and commented via duplicate issue S-174:

The finding and its duplicates outline that the accumulated operator fee is factored in total asset calculations despite being meant to be redeemed as a fee.

Apart from contradicting the EIP-4626 standard, it allows the operator fee to be redeemed by users, undervalues deposits made when a non-zero operator fee exists, and abruptly reduces the total assets whenever the operator fee is claimed.

I believe the consistent dilution of all incoming deposits whenever a non-zero operator fee is present to be a significant issue and one that would merit a high severity rating. Specifically:

- The vulnerability is consistently present whenever an operator fee is realized (i.e. `operatorFeeAmount` is non-zero) - Likelihood of High.
- The impact of the vulnerability is significant as it devalues all incoming user deposits whenever a non-zero fee is present and can also result in the `operatorFeeAmount` becoming irredeemable in extreme circumstances (i.e. total withdrawal of vault) - Impact of Medium.

Combining the likelihood and impact traits above, I believe that a severity level of high is better suited for this issue.

Liquid Ron mitigated:

Add `operatorFeeAmount` in `totalAssets` calculations.

Status: Mitigation confirmed. Full details in reports from Aamir, Orpse, and ilchovski.

Medium Risk Findings (2)

[M-01] User can earn rewards by frontrunning the new rewards accumulation in Ron staking without actually delegating his tokens

Submitted by Aamir, also found by Orpse, OxOdd, Oxrex, gesha17, ilchovski, KupiaSec, and spuriousdragon

LiquidProxy.sol#L39

Finding Description and Impact

The Ron staking contract let us earn rewards by delegating our tokens to a validator. But you will only earn rewards on the lowest balance of the day (source). So if you delegate your tokens on the first day, you are going to earn 0 rewards for that day as your lowest balance was 0 on that day. This will happens with every new delegator.

Now the issue with `LiquidRon` is that, there will be many users who will be depositing their tokens in it. And there is no such kind of time or amount restriction for new delegators if some people have already delegated before them. So with direct delegation, the new rewards flow will be this:

```
User -> delegate -> RonStaking -> Wait atleast a day -> New Rewards
```

But if we deposit through `LiquidRon` it has become this:

```
User -> LiquidRon -> LiquidProxy -> New Rewards
```

Now a user can earn rewards by just depositing the tokens into the `LiquidRon` by frontrunning the new rewards arrival and immediately withdraw them. But if a user try to do this by frontrunning the `LiquidRon::harvest(...)` call, this will not be possible. Because when he deposits, he will get shares in return which will already be accounting for any unclaimed rewards through `getTotalRewards(...)` in `LiqiuidRon::totalAssets(...)`:

```
    function totalAssets() public view override returns (uint256) {
@>        return super.totalAssets() + getTotalStaked() +
getTotalRewards();
    }
```

But instead of frontrunning this `harvest(...)` call, a user can just frontrun the new rewards arrival in the `RonStaking` contract itself. Because as per the Ronin staking docs, a user will be able to claim new rewards after every 24 hours.

Also, if we look at the `_getReward(...)` (used by claimRewards etc.) function of the Ronin staking contract, the rewards will be same as before as long as we are not in new period:

```
  function _getReward(
    address poolId,
    address user,
    uint256 latestPeriod,
    uint256 latestStakingAmount
  ) internal view returns (uint256) {
    UserRewardFields storage _reward = _userReward[poolId][user];
```

```
@>    if (_reward.lastPeriod == latestPeriod) {
@>      return _reward.debited;
    }
```

Github: Link

So if a user frontrun before this, the `getTotalRewards(...)` function will also not account for new rewards as long as we are not in new period.

Also note that, if a user frontruns and deposit, he didn't not actually delegated his tokens as the tokens are only delegated once the operator calls `LiquidRon::delegateAmount(...)` function:

```
    function delegateAmount(uint256 _proxyIndex, uint256[] calldata
_amounts, address[] calldata _consensusAddrs)
        external
        onlyOperator
        whenNotPaused
    {
        address stakingProxy = stakingProxies[_proxyIndex];
        uint256 total;

        if (stakingProxy == address(0)) revert ErrBadProxy();
        // @audit how many max validators can be there?
        for (uint256 i = 0; i < _amounts.length; i++) {
            if (_amounts[i] == 0) revert ErrNotZero();
            _tryPushValidator(_consensusAddrs[i]);
            total += _amounts[i];
        }
@>        _withdrawRONTo(stakingProxy, total);
@>        ILiquidProxy(stakingProxy).delegateAmount(_amounts,
_consensusAddrs);
    }
```

So a user is just depositing, claiming and withdrawing his tokens but not actually delegating.

The following attack can be summarized like this:

1. There are currently millions of delegated tokens deposited into the `LiquidRon` which are delegated to many validators through `LiquidProxy`. And each day the `LiquidRon` is earning let's 1000 tokens as a reward.
2. But these rewards are only claimable in the new epoch.
3. So a user keeps an eye on the new rewards arrival into the RonStaking for the `LiquidRon` and also on the new epoch update.

4. Once rewards are arrived he waits for the epoch update, and frontruns this epoch update and deposits into the `LiquidRon`.
5. Once this new epoch is updated, these new rewards will be starting to reflect into the the share price through `LiquidRon::totalAssets(...)`.
6. User withdraws immediately with the profit.

In the above case, the user didn't have to stake his tokens and still earns the share of the rewards. Also, this can be done each and every time to claim the rewards.

He also didn't have to bear any risk of loss in case something bad happens (maybe some kind of attack) and the share price value goes down; while others will also have to bear this loss.

Recommended Mitigation Steps

Incorporate some kind of locking mechanism for new depositor like Ron staking contract does. Or, go for some alternate option.

Owl (Liquid Ron) confirmed and commented:

Vampire attack as I like to call it.

Valid finding, definitely need to think of a viable mitigation solution. For now, adding a deposit fee seems the best solution I have found (deposit fee would be equivalent to 1 day worth of reward expected based on amount deposited).

0xsomeone (judge) commented:

The finding and its duplicates outline a valid attack path via which users are able to arbitrage upcoming reward updates to capture the rewards without actually staking their assets for a long period of time in the system.

I believe a medium-severity rating is justifiable, and mitigation for the vulnerability outlined will be hard to implement properly unless a simplistic measure such as a deposit fee is implemented. Alternatives would be to finalize deposits via operators or permit deposits to occur within a predefined time window of a reward issuance, ensuring new deposits are finalized after a reward update has occurred recently.

Liquid Ron mitigated:

Add a deposit fee that can be reset every period based on daily expected rewards.

Status: Mitigation confirmed. Full details in report from Aamir.

[M-02] Operators are unable to perform any actions due to incorrect modifier implementation

Submitted by zanderbyte, also found by 056Security, Orpse, 0x0107, 0x0bserver, 0x0dd, 0x11singh99, 0x23r0, 0xAadi, 0xAkira, 0xAlipede, 0xAsen, 0xastronatey, 0xauditagent, 0xaudron, 0xAura, 0xfocusNode, 0xhuh2005, 0xiehnnkta, 0xLeveler, 0xMosh,

0xmujahid002, *0xnolo*, *0xodus*, *0xRajkumar*, *0xrex*, *0xterrah*, *0xvd*, *13u9*, *4rdiii*, *Aamir*, *aariiif*, *Adotsam*, *Afriauditor*, *Agontuk*, *air_0x*, *Albort*, *anonymousjoe*, *arman*, *aua_oo7*, *AvantGard*, *axelot*, *backboard9654*, *BanditxOx*, *bani*, *Bauchibred*, *Bbash*, *Bluedragon101*, *brevis*, *BRONZEDISC*, *BroRUok*, *Bz*, *ccvascocc*, *ChainSentry*, *Cli7max*, *CrazyMoose*, *crmx_lom*, *csanuragjain*, *cylzxje*, *CyXq*, *d4r3d3v1l*, *Daniel_eth*, *Daniel526*, *dd0x7e8*, *debo*, *Dec3mber*, *DemoreX*, *den-sosnowsky*, *dexcripter*, *DharkArtz*, *DoD4uFN*, *dreamcoder*, *edwardmintel*, *ElCid*, *elolpuer*, *eLSeR17*, *EPSec*, *erictee*, *eternal1328*, *ewwrekt*, *Falendar*, *federodes*, *Flare*, *francoHacker*, *fromeo_016*, *gesha17*, *gregom*, *gxh191*, *HardlyDifficult*, *harry_cryptodev*, *HChang26*, *holydevoti0n*, *honey-k12*, *hrmneffdii*, *hyuunn*, *IceBear*, *ilchovski*, *importDev*, *Incogknito*, *inh3l*, *interestingTimes*, *itsabinashb*, *JakeFromStateFarm*, *JCN*, *jesusrod15*, *jkk812812*, *Josh4324*, *ka14ar*, *Kalogerone*, *Kek*, *King_9aimon*, *klau5*, *KupiaSec*, *kutugu*, *Lamsy*, *levi_104*, *linemi*, *LLakterian*, *LonelyWolfDemon*, *Maglov*, *mahdifa*, *Mahi_Vasisth*, *marwen*, *miaowu*, *mtberi*, *muellerberndt*, *MukulKolpe*, *muncentu*, *mxteem*, *N0nce*, *newspacexyz*, *NexusAudits*, *nnez*, *noured23*, *oakcobalt*, *octeezy*, *ok567*, *Olugbenga*, *oxelmiguel12*, *p_crypt0*, *peanuts*, *persik228*, *pfapostol*, *phoenixV110*, *Prestige*, *pulse*, *PumpkingWok*, *Punith*, *queen*, *RaOne*, *rare_one*, *rbserver*, *Rhaydden*, *Riceee*, *rishab*, *robertauditor*, *Rolando*, *rudhra*, *Ryonen*, *sabanaku77*, *safie*, *sajeevan_58356*, *Samueltroydomi*, *seerether*, *serial-coder*, *Shahil_Hussain*, *shaka*, *Shinobi*, *Shipkata494*, *silver_eth*, *sl1*, *SmartAuditPro*, *sohrabhind*, *Sparrow*, *spuriousdragon*, *stuart_the_minion*, *t0x1c*, *tachida2k*, *Tadev*, *Tamer*, *teoslaf*, *Tiannah*, *Tigerfrake*, *Timeless*, *tonitonitoni*, *Topmark*, *tpiliposian*, *Trepid*, *typicalHuman*, *udo*, *udogodwin*, *Udsen*, *unique*, *unnamed*, *vahdrak1*, *verboten_viking*, *victortheoracle*, *vladi319*, *vulkan_xx*, *wellbyt3*, *Xcrypt*, *XDZIBECX*, *YouCrossTheLineAlfie*, *Z-bra*, *zraxx*, *zubyoz*, and *ZZhelev*

LiquidRon.sol#L91

Finding Description and Impact

The `onlyOperator` modifier in `LiquidRon` contract is intended to restrict access to specific functions to either the `owner` or `operator`. Functions like `harvest`, `delegateAmount`, and `harvestAndDelegateRewards` rely on this modifier for access control.

However, the modifier implementation is incorrect and will always revert when called by any address other than the owner, even if the caller is a valid operator. As a result, operators are completely unable to perform any of their intended actions.

As we can see, if `msg.sender` is not the owner, the first condition evaluates to true, triggering a revert. Even if we ignore the first condition, when an operator calls a function using this modifier, `operator[msg.sender]` evaluates to true, causing the revert to be triggered again.

```
    /// @dev Modifier to restrict access of a function to an operator or
owner
    modifier onlyOperator() {
```

```
        if (msg.sender != owner() || operator[msg.sender]) revert
ErrInvalidOperator();
        _;
    }
```

Proof of Concept

Paste the following test in `LquidRon.operator.t.sol`:

```
    function test_wronModifierIImplementation() public {
        address operator = makeAddr("operator");
        liquidRon.updateOperator(operator, true);

        assertTrue(liquidRon.operator(operator));

        vm.startPrank(operator);
        vm.expectRevert(LiquidRon.ErrInvalidOperator.selector);
        liquidRon.harvest(1, consensusAddrs);
    }
```

Recommended Mitigation Steps

The modifier should revert only if the `msg.sender` is neither the owner nor an operator:

```
    /// @dev Modifier to restrict access of a function to an operator or
owner
    modifier onlyOperator() {
-       if (msg.sender != owner() || operator[msg.sender]) revert
ErrInvalidOperator();
+       if (msg.sender != owner() && !operator[msg.sender]) revert
ErrInvalidOperator();
        _;
    }
```

Owl (Liquid Ron) confirmed and commented:

I agree that the finding is valid, I simply wonder if the severity is justified.

The modifier doesn't permit for anyone to call the operator functions, which for me is a high severity. With this reasoning, the finding could be of medium severity as workarounds external to the contract are possible (owner is a contract with actual correct usage of the operator finding if such code was live).

Oxsomeone (judge) decreased severity to Medium and commented:

The submission details an incorrect code implementation that effectively increases the level of access control for functions that utilize the `onlyOperator` modifier as operators are not able to access those functions and the owner is the only one capable of invoking them.

I believe that such a finding does not result in a material vulnerability and instead falls under the "self-evident code mistake" clause of the relevant Supreme Court verdicts. In such instances, mistakes in code that are clear yet do not result in a significant vulnerability would be considered medium severity and I consider this approach to apply here, thus downgrading the issue to medium severity.

Liquid Ron mitigated:

Bad operator modifer.

Status: Mitigation confirmed. Full details in reports from Aamir, Orpse, and ilchovski.

Low Risk and Non-Critical Issues

For this audit, 17 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **IIIIHunterIIII** received the top score from the judge.

The following wardens also submitted reports: 0x23r0, OxAadi, Oxodus, Oxrex, atoko, Bauchibred, Bigsam, Daniel526, inh3l, K42, Kalogerone, magiccentaur, pfapostol, rbserver, Rhaydden, and Sparrow.

Summary

ID	TITLE
[01]	Risk `Out-Of-Gas` reverts during deposit and withdraw in reasonable circumstances
[04]	Some users can deposit but can never withdraw

Note: findings `02` (redundant with H-01) and `03` (judged invalid) from IIIIHunterIIII's submission have been omitted from this section of the report. The original numbering has been kept for ease of reference.

[01] Risk `Out-Of-Gas` reverts during deposit and withdraw in reasonable circumstances

The `totalAssets()` function in `LiquidRon.sol` iterates over all staking proxies and consensus addresses to calculate the total assets controlled by the contract. This function is called during critical operations such as deposits and withdrawals. However, if the number of

staking proxies or consensus addresses is large, the function can consume excessive gas, potentially exceeding the Ethereum block gas limit (30M gas). This can lead to out-of-gas (OOG) reverts, rendering the contract unusable for deposits and withdrawals in high-scale scenarios.

The most reasonable numbers I could come across are 60 validator and 100 staking proxy deployed:

- While this seems large:
  - The nature of staking protocols usually involve more than 100 validators.
  - If the TVL of the LiquidRonin is increasing and multiple user interactions are happening daily, they will need to deploy more proxies.
- The above two points make the bug more likely to rise.

Impact

- **Denial of Service (DoS):** If `totalAssets()` reverts due to OOG, users will be unable to deposit or withdraw funds, effectively freezing the contract temporarily till the operator claim and undelegate from number of operators to delete some them to decrease the iteration numbers on consensus adresses.
- **Scalability issues:** The contract cannot handle a large number of staking proxies or consensus addresses, limiting its scalability.
- **User funds at risk:** If withdrawals are blocked due to OOG reverts, users may be unable to access their funds. (same as first point).

Proof of Concept

Paste this in `LiquidRon.t.sol`:

```
    function test_totalAssets_OOG() public {
    // Deploy multiple staking proxies
    uint256 proxyCount = 100; // Adjust this number to test different
scenarios
    for (uint256 i = 0; i < proxyCount; i++) {
        liquidRon.deployStakingProxy();
    }

    // Add a large number of consensus addresses
    uint256 consensusCount = 60; // Adjust this number to test different
scenarios
    address[] memory consensusAddrs = new address[](consensusCount);
    for (uint256 i = 0; i < consensusCount; i++) {
        consensusAddrs[i] = address(uint160(i + 1)); // Generate unique
addresses
```

```
    }

    // Deposit some RON to initialize the system
    uint256 depositAmount = 1000000000000000000000000000000 ether;

    deal(address(this), depositAmount * 10);

    liquidRon.deposit{value: depositAmount * 10}();

    // Delegate amounts to consensus addresses
    uint256[] memory amounts = new uint256[](consensusCount);
    for (uint256 i = 0; i < consensusCount; i++) {
        amounts[i] = 1;
    }
    for (uint256 i = 0; i < proxyCount; i++) {
        liquidRon.delegateAmount(i, amounts, consensusAddrs);
    }

    // Call totalAssets() and check for OOG reverts
    uint256 blockGasLimit = 30_000_000;
    uint256 totalAssets;
    // passing the block gas limit as a parameter to simulate a real
environment block limit
    try liquidRon.totalAssets{gas: blockGasLimit}() returns (uint256
_totalAssets) {
        totalAssets = _totalAssets;
    } catch {
        revert("OOG in totalAssets()");
    }

    // Assert that totalAssets is greater than 0
    assertTrue(totalAssets > 0, "totalAssets should be greater than 0");
}
```

The test fails with an `OutOfGas` error, demonstrating that `totalAssets()` consumes excessive gas and reverts when the number of staking proxies and consensus addresses is large.

Recommended Mitigation Steps

1. **Optimize `totalAssets()` function:**

- **Cache results:** Cache the results of expensive operations (e.g., staked amounts and rewards) to avoid recalculating them on every call.
- **Batch processing:** Process staking proxies and consensus addresses in batches to reduce gas consumption per transaction.

- **Off-chain calculation**: Use an off-chain service to calculate total assets and provide the result to the contract via a trusted oracle.

2. **Limit the number of proxies and consensus addresses**:
- **Enforce limits**: Set a maximum limit on the number of staking proxies and consensus addresses that can be added to the contract.
- **Prune inactive addresses**: Regularly prune inactive consensus addresses to reduce the number of iterations in `totalAssets()`.

[04] Some users can deposit but can never withdraw

In `LiquidRon`, the contract allow users to `deposit()` or `mint()` using the `wRON` token, but during `withdraw()` or `burn()` users are forced to receive native `RON` or the txn will revert:

- This is a problem cause the depositor may have been a contract that doesn't implement `receive()` or payable functions.
- The depositor may have been a protocol that is building on top of `liquidRon` too.

This makes the above two cases to never be able to get back any tokens and have their funds stuck.

Proof of Concept

1. A contract not having receive or payable functions deposit in `liquidRon` using `wRON`.
2. Time passes and he wants to withdraw.
3. Any call to `withdraw()` or `redeem()` will revert during `_withdrawRONTo()` call.

```
File: RonHelper.sol
38:     function _withdrawRONTo(address to, uint256 amount) internal {
39:         IWRON(wron).withdraw(amount);
40:         (bool success, ) = to.call{value: amount}("");
41:         if (!success) revert ErrWithdrawFailed();
42:     }
```

4. Funds are stuck forever.

Recommended Mitigation Steps

Wrap the native call that if failed, wrap the native tokens and send them to the receiver as `wRON`.

Mitigation Review

Introduction

Following the C4 audit, 3 wardens (Aamir, ilchovski, and Orpse) reviewed the mitigations implemented by the Liquid Ron team. Additional details can be found within the C4 Liquid Ron Mitigation Review repository.

Mitigation Review Scope

Branch

- Branch: https://github.com/OwlOfMoistness/liquid_ron/tree/ca4-mitigation
- Commits: https://github.com/OwlOfMoistness/liquid_ron/compare/main...ca4-mitigation

Mitigation of High & Medium Severity Issues

MITIGATION	MITIGATION OF	PURPOSE
Link	H-01 aka F-3	Add operatorFeeAmount in totalAssets calculations
Link	M-01 aka F-10	Add a deposit fee that can be reset every period based on daily expected rewards
Link	M-02 aka F-23	Bad operator modifer

Additional Scope to be Reviewed

These are additional changes that were in scope.

MITIGATION	REF-ID	PURPOSE
Link	FX-1*	Update flow of withdrawal to add changeable receiver
Link	F-25	Replace validator data storage from consensus addresses to IDs which never change
Link	F-2	Add start index to start loop on specif validator and length of computation

MITIGATION	REF-ID	PURPOSE
Link	F-45	QA, remove unused mapping
Link	F-32	Fix wrong event emission
Link	F-156	Clear validator Index when removing it
Link	F-22	Remove for loop
Link	F-1	Prevent native deposits when paused
Link	S-736: Low-4 **	Improve getTotalStaked() to prevent recomputing state each call by tracking internally
Link 1, Link 2	ADD-01	QAs: getValidator func, payable withdraw ron, remove _checkIfPaused, immutable proxy var, check src/dst in proxy, remove dead code, deposit payable has receiver param
Link	ADD-02	Fix test
Link	ADD-03	Add forge lib
Link	ADD-04	Add periodStartVariable for external data tracking

*`FX-1` *represents multiple findings from the initial audit: F-18, F-17, and F-27. These have all been fixed by omitting the* `_checkUserCanReceiveRon` *implementation and replacing it with the capability to specify a different* `receiver` *when performing withdrawal requests as well as deposits.*

***Commit title incorrectly mentions* `S-726`*.*

Mitigation Review Summary

During the review, all in-scope mitigations were confirmed, and no new high- or medium-risk issues were discovered by the wardens. The table below provides further detail.

ORIGINAL ISSUE	STATUS	FULL DETAILS
H-01 (F-3)	🟢 Mitigation Confirmed	Reports from Aamir, Orpse, and ilchovski

ORIGINAL ISSUE	STATUS	FULL DETAILS
M-01 (F-10)	🟢 Mitigation Confirmed	Report from Aamir
M-02 (F-23)	🟢 Mitigation Confirmed	Reports from Aamir, Orpse, and ilchovski
FX-1	🟢 Mitigation Confirmed	Reports from Orpse, ilchovski, and Aamir
F-25	🟢 Mitigation Confirmed	Reports from Orpse, ilchovski, and Aamir
F-2	🟢 Mitigation Confirmed	Reports from Orpse, ilchovski, and Aamir
F-45	🟢 Mitigation Confirmed	Reports from Orpse, ilchovski, and Aamir
F-32	🟢 Mitigation Confirmed	Reports from Orpse, ilchovski, and Aamir
F-156	🟢 Mitigation Confirmed	Reports from Aamir, Orpse, and ilchovski
F-22	🟢 Mitigation Confirmed	Reports from Orpse, ilchovski, and Aamir
F-1	🟢 Mitigation Confirmed	Reports from Orpse, ilchovski, and Aamir
S-736: Low-4	🟢 Mitigation Confirmed	Reports from Orpse, ilchovski, and Aamir
ADD-01	🟢 Mitigation Confirmed	Reports from Orpse, Aamir, and ilchovski
ADD-02	🟢 Mitigation Confirmed	Reports from Aamir, Orpse, and ilchovski
ADD-03	🟢 Mitigation Confirmed	Reports from Orpse, Aamir, and ilchovski
ADD-04	🟢 Mitigation Confirmed	Reports from ilchovski, Aamir, and Orpse

Disclosures

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

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

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.