SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public Best Efforts
Prepared for:	Zivoe
Prepared by:	Sherlock
Lead Security Expert:	cergyk
Dates Audited:	April 8 - April 25, 2024
Prepared on:	June 6, 2024

Introduction

Zivoe is a real-world asset credit protocol aiming to disrupt predatory high-interest consumer lending. Leveraging a B2B2C model, Zivoe offers on-chain loans to regulated consumer lending entities, who then use that capital to fund consumer credit products off-chain. These entities then utilize the yield from such products to fulfill their on-chain obligations to Zivoe.

Scope

Repository: Zivoe/zivoe-core-foundry

Branch: master

Commit: ad27cffdf96eaaa33274bfba0dda9b60e36d29a2

Repository: Zivoe/zivoe-core-testing

Branch: main

Commit: 478e13327af672e7b5dfffda38b69acfed37b346

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

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

Issues found

Medium	High
15	7

SHERLOCK

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

cergyk
BoRonGod
0x73696d616f
BiasedMerc
Ironsidesec
AMOW
SilverChariot
0xpiken
Drynooo
AllTooWell
recursiveEth
0xBhumii
ether_sky
den_sosnovskyi
lemonmon
t0x1c
amar
denzi_
dany.armstrong90
Nihavent
Tendency
jasonxiale
JigglypuffAndPikachu
KupiaSec
dimulski
KingNFT
SUPERMAN_I4G
pseudoArtist
saidam017
Audinarey
Maniacs
samuraii77
joicygiore
rbserver
beWater0given
Krace
Quanta
flacko
Tricko
sl1
marchev
y4y
ZanyBonzy
thank_you
Bauchibred
0xvj
mt030d
9oelm
DPS
0xAnmol
Ruhum
Afriaudit
sakshamguruji
zarkk01
Varun_05
blockchain555
14si2o_Flint
Tychai0s
coffiasd
didi
hulkvision
0xShax2nk.in
CL001
Dliteofficial
araj
forgebyola
asui
supercool
Shield
Aymen0909
FastTiger
krikolkk
0xboriskataa
Kunhah
Bauer
Timenov
pashap9990
0brxce
novaman33
heedfxn
sunill_eth
0rpse
blackhole

Issue H-1: Anyone could call `depositReward` with zero reward to extend the period finish time

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/11

Found by

0brxce, 0xAnmol, 0xboriskataa, 0xpiken, 0xvj, 14si2o_Flint, 9oelm, AMOW, Afriaudit, Bauer, CL001, Dliteofficial, Drynooo, FastTiger, Ironsidesec, Krace, Kunhah, Maniacs, Nihavent, Ruhum, SilverChariot, Timenov, Tychai0s, amar, araj, asui, blockchain555, cergyk, coffiasd, dany.armstrong90, dimulski, forgebyola, heedfxn, jasonxiale, joicygiore, krikolkk, lemonmon, marchev, mt030d, novaman33, pashap9990, rbserver, sakshamguruji, sl1, sunill_eth, t0x1c

Summary

Anyone could extend the reward finish time, potentially resulting in users receiving fewer rewards than expected within the same time period.

Vulnerability Detail

The function `depositReward` can be called by anyone, even with zero rewards, allowing it to be exploited to extend the reward finish time at little cost. This could result in loss of rewards; for instance, if there are 10 DAI rewards within a 10-day period, a malicious user could extend the finish time on *day 5*, extending the finish time to the 15th day. Participants would only receive 7.5 DAI by the 10th day.

```
function depositReward(address _rewardsToken, uint256 reward) external
↪  updateReward(address(0)) nonReentrant {
    IERC20(_rewardsToken).safeTransferFrom(_msgSender(), address(this), reward);

    // Update vesting accounting for reward (if existing rewards being
↪  distributed, increase proportionally).
    if (block.timestamp >= rewardData[_rewardsToken].periodFinish) {
        rewardData[_rewardsToken].rewardRate =
↪  reward.div(rewardData[_rewardsToken].rewardsDuration);
    } else {
        uint256 remaining =
↪  rewardData[_rewardsToken].periodFinish.sub(block.timestamp);
        uint256 leftover = remaining.mul(rewardData[_rewardsToken].rewardRate);
        rewardData[_rewardsToken].rewardRate =
↪  reward.add(leftover).div(rewardData[_rewardsToken].rewardsDuration);
    }
    rewardData[_rewardsToken].lastUpdateTime = block.timestamp;
    rewardData[_rewardsToken].periodFinish =
 ↪  block.timestamp.add(rewardData[_rewardsToken].rewardsDuration);
    emit RewardDeposited(_rewardsToken, reward, _msgSender());
}
```

POC

Add the test to `zivoe-core-testing/src/TESTS_Core/Test_ZivoeRewards.sol` and run it with `forge test --match-test test_ZivoeRewards_deposit_zero --rpc-url <RPC_URL_MAINNET>`

```
diff --git a/zivoe-core-testing/src/TESTS_Core/Test_ZivoeRewards.sol
 ↪  b/zivoe-core-testing/src/TESTS_Core/Test_ZivoeRewards.sol
index f5353b6..870a531 100644
--- a/zivoe-core-testing/src/TESTS_Core/Test_ZivoeRewards.sol
+++ b/zivoe-core-testing/src/TESTS_Core/Test_ZivoeRewards.sol
@@ -685,6 +685,33 @@ contract Test_ZivoeRewards is Utility {

     }

+    function test_ZivoeRewards_deposit_zero() public {
+
+        depositReward_DAI(address(stZVE), 1);
+
+        (
+            uint256 rewardsDuration,
+            uint256 _prePeriodFinish,
+            uint256 _preRewardRate,
+            uint256 lastUpdateTime,
+            uint256 rewardPerTokenStored
+        ) = stZVE.rewardData(DAI);
+        console.log("period finish ", _prePeriodFinish);
+
+        vm.warp(block.timestamp + 1 days);
+
+        depositReward_DAI(address(stZVE), 0);
+
+        (,
+            uint256 _afterPeriodFinish,
+            ,
+            ,
+        ) = stZVE.rewardData(DAI);
+        console.log("period finish ", _afterPeriodFinish);
+        //  extend the Finish 1 day
+        assertEq(_afterPeriodFinish - _prePeriodFinish, 1 days);
+     }
+
      function test_ZivoeRewards_getRewards_works(uint96 random) public {

          uint256 deposit = uint256(random) + 100 ether; // Minimum 100 DAI
↪   deposit.
```

Impact

Anyonce could extend the reward finish time and the users may receive less rewards than expected during the same time period.

Code Snippet

https://github.com/sherlock-audit/2024-03-zivoe/blob/d4111645b19a1ad3ccc899bea073b6f19be04ccd/zivoe-core-foundry/src/ZivoeRewards.sol#L228-L243

Tool used

Foundry

Recommendation

Only specific users are allowed to call function `depositReward`

Discussion

pseudonaut

Valid, considering adding whitelist

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> high, allows anyone to extend reward finish time indefinitely and decrease reward rate.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Zivoe/zivoe-core-foundry/pull/260

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-2: ITO can be manipulated

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/12

Found by

0xShax2nk.in, 0xpiken, 0xvj, 14si2o_Flint, 9oelm, AMOW, Drynooo, Ironsidesec, JigglypuffAndPikachu, KupiaSec, Nihavent, SilverChariot, Tendency, blockchain555, coffiasd, dany.armstrong90, didi, ether_sky, hulkvision, joicygiore, lemonmon, mt030d, saidam017, sl1, zarkk01

Summary

The ITO allocates 3 pZVE tokens per senior token minted and 1 pZVE token per junior token minted. When the offering period ends, users can claim the protocol `ZVE token` depending on the share of all pZVE they hold. Only 5% of the total ZVE tokens will be distributed to users, which is equal to `1.25M` tokens.

The ITO can be manipulated because it uses `totalSupply()` in its calculations.

Vulnerability Detail

ZivoeITO.claimAirdrop() calculates the amount of ZVE tokens that should be vested to a certain user. It then creates a vesting schedule and sends all junior and senior tokens to their recipient.

The formula is $pZVEOwned/pZVETotal * totalZVEAirdropped$ (in the code these are called `upper`, `middle` and `lower`).

```
uint256 upper = seniorCreditsOwned + juniorCreditsOwned;
uint256 middle = IERC20(IZivoeGlobals_ITO(GBL).ZVE()).totalSupply() / 20;
uint256 lower = IERC20(IZivoeGlobals_ITO(GBL).zSTT()).totalSupply() * 3 + (
    IERC20(IZivoeGlobals_ITO(GBL).zJTT()).totalSupply()
);
```

These calculations can be manipulated because they use `totalSupply()`. The tranche tokens have a public burn() function.

An attacker can use 2 accounts to enter the ITO. They will deposit large amounts of stablecoins towards the senior tranche. When the airdrop starts, they can claim their senior tokens and start vesting ZVE tokens. The senior tokens can then be burned. Now, when the attacker calls the `claimAirdrop` function with their second account, the denominator of the above equation will be much smaller, allowing them to claim much more ZVE tokens than they are entitled to.

Impact

There are 2 impacts from exploiting this vulnerability:

- a malicious entity can claim excessively large part of the airdrop and gain governance power in the protocol
- since the attacker would have gained unexpectedly large amount of ZVE tokens and the total ZVE to be distributed will be `1.25M`, the users that claim after the attacker may not be able to do so if the amount they are entitled to, added to the stolen ZVE, exceeds `1.25M`.

Code Snippet

Add this function to Test_ZivoeITO.sol and import the `console`.

You can comment the line where Sue burns their tokens and see the differences in the logs.

```
function test_StealZVE() public {
    // Sam is an honest actor, while Bob is a malicious one
    mint("DAI", address(sam), 3_000_000 ether);
    mint("DAI", address(bob), 2_000_000 ether);
    zvl.try_commence(address(ITO));

    // Bob has another Ethereum account, Sue
    bob.try_transferToken(DAI, address(sue), 1_000_000 ether);

    // give approvals
    assert(sam.try_approveToken(DAI, address(ITO), type(uint256).max));
    assert(bob.try_approveToken(DAI, address(ITO), type(uint256).max));
    assert(sue.try_approveToken(DAI, address(ITO), type(uint256).max));

    // Sam deposits 2M DAI to senior tranche and 400k to the junior one
    hevm.prank(address(sam));
    ITO.depositBoth(2_000_000 ether, DAI, 400_000, DAI);

    // Bob deposits 2M DAI into the senior tranche using his both accounts
    hevm.prank(address(bob));
    ITO.depositSenior(1_000_000 ether, DAI);

    hevm.prank(address(sue));
    ITO.depositSenior(1_000_000 ether, DAI);

    // Move the timestamp after the end of the ITO
    hevm.warp(block.timestamp + 31 days);

    ITO.claimAirdrop(address(sue));
    (, , , uint256 totalVesting, , , ) = vestZVE.viewSchedule(address(sue));

    // Sue burn all senior tokens
    vm.prank(address(sue));
    zSTT.burn(1_000_000 ether);

    console.log('Sue vesting: ', totalVesting / 1e18);

    ITO.claimAirdrop(address(bob));
    (, , , totalVesting, , , ) = vestZVE.viewSchedule(address(bob));

    console.log('Bob vesting: ', totalVesting / 1e18);

    ITO.claimAirdrop(address(sam));
    (, , , totalVesting, , , ) = vestZVE.viewSchedule(address(sam));

    console.log('Sam vesting: ', totalVesting / 1e18);
}
```

Fair vesting without prior burning

```
Sue vesting:  312499
Bob vesting:  312499
Sam vesting:  625001
```

Vesting after burning

```
Sue vesting:  312499
Bob vesting:  416666
Sam vesting:  833333
```

Bob and Sue will be able to claim `~750 000` ZVE tokens and Sam will not be able to claim any, because the total exceeds `1.25M`.

Tool used

Manual Review

Recommendation

Introduce a few new variables in the ITO contract.

```
bool hasAirdropped;
uint256 totalZVE;
```

```
uint256 totalzSTT;
uint256 totalzJTT;
```

Then check if the call to `claimAidrop` is a first one and if it is, initialize the variables. Use these variables in the vesting calculations.

```
function claimAirdrop(address depositor) external returns (
    uint256 zSTTClaimed, uint256 zJTTClaimed, uint256 ZVEVested
) {
        ...
        if (!hasAirdropped) {
              totalZVE = IERC20(IZivoeGlobals_ITO(GBL).ZVE()).totalSupply();
              totalzSTT = IERC20(IZivoeGlobals_ITO(GBL).zSTT()).totalSupply();
              totalzJTT = IERC20(IZivoeGlobals_ITO(GBL).zJTT()).totalSupply();
              hasAirdropped = true;
        }
        ...

       uint256 upper = seniorCreditsOwned + juniorCreditsOwned;
       uint256 middle = totalZVE / 20;
       uint256 lower = totalzSTT * 3 + totalzJTT;
  }
```

Discussion

pseudonaut

This is interesting - technically we can support overflow in these situations because there's more ZVE available in other contract. User's would need to burn important tokens, and lock-up capital for extended period of time (more than 1 block) and upside isn't present against what rewards are and capital-loss - net-positive for protocol

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> high, allows to inflate ITO vesting amounts and cause loss of funds from ITO for the other users

panprog

Many dups of this issue talk about naturally increasing `totalSupply` due to new deposits, which decreases airdrop for users who start vesting later. The core reason is the same.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Zivoe/zivoe-core-foundry/pull/266

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-3: User cannot withdraw stakingToken due to incorrect calculation of _totalSupply

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/31

Found by

0xAnmol, 0xvj, 9oelm, AMOW, Afriaudit, Audinarey, Aymen0909, CL001, Dliteofficial, Drynooo, Ironsidesec, JigglypuffAndPikachu, KingNFT, Ruhum, Shield, SilverChariot, Tendency, Varun_05, amar, araj, asui, beWater0given, denzi_, dimulski, forgebyola, lemonmon, marchev, mt030d, rbserver, saidam017, sakshamguruji, samuraii77, sl1, supercool, t0x1c, zarkk01

Summary

User cannot withdraw `stakingToken` due to incorrect calculation of `_totalSupply`

Vulnerability Detail

Given:

1.userA VestingSchedule `amountToVest`= 20 ether ,`daysToCliff`=30 , `daysToVest`=120

2.userB VestingSchedule `amountToVest`= 30 ether , `daysToCliff`=30, `daysToVest`=120 _totalSupply =20 +30= 50

3.skip 60 days userB withdraw (`amountWithdrawable =15`) _totalSupply = 50-15=35

4.`revoke` userB `VestingSchedule` _totalSupply = 35-30=5 https://github.com/sherlock-audit/2024-03-zivoe/blob/d4111645b19a1ad3ccc899bea073b6f19be04ccd/zivoe-core-foundry/src/ZivoeRewardsVesting.sol#L451

5.skip 60 days (120 days later) userA withdraw `(amountWithdrawable =20)` _totalSupply = 5-20 lead to revert https://github.com/sherlock-audit/2024-03-zivoe/blob/d4111645b19a1ad3ccc899bea073b6f19be04ccd/zivoe-core-foundry/src/ZivoeRewardsVesting.sol#L508

POC: src/TESTS_Core/Test_ZivoeRewardsVesting.sol

```
    function testpoc() public{
        assert(zvl.try_createVestingSchedule(
            address(vestZVE),
            address(qcp),
            30,
            120,
            20 ether,
            true
        ));
        assert(zvl.try_createVestingSchedule(
            address(vestZVE),
            address(pam),
            30,
            120,
            30 ether,
            true
        ));


        hevm.warp(block.timestamp + 60 days);

        vestZVE.amountWithdrawable(address(pam));
        //pam withdraw
         hevm.startPrank(address(pam));
        vestZVE.withdraw();
        hevm.stopPrank();
        vestZVE.totalSupply();

        //revokeVesting
        hevm.startPrank(address(zvl));
        vestZVE.revokeVestingSchedule(address(pam));
        hevm.stopPrank();
        vestZVE.totalSupply();

        //qcp withdraw
        hevm.warp(block.timestamp + 60 days);
        vestZVE.amountWithdrawable(address(qcp));
         hevm.startPrank(address(qcp));
        vestZVE.withdraw();
        hevm.stopPrank();
    }
[FAIL. Reason: panic: arithmetic underflow or overflow (0x11)] testpoc() (gas:
↪   795816)
```

Impact

User cannot withdraw stakingToken

Code Snippet

https://github.com/sherlock-audit/2024-03-zivoe/blob/d4111645b19a1ad3ccc899bea073b6f19be04ccd/zivoe-core-foundry/src/ZivoeRewardsVesting.sol#L451

Tool used

Manual Review

Recommendation

```
function revokeVestingSchedule(address account) external updateReward(account)
↪  onlyZVLOrITO nonReentrant {
     require(
         vestingScheduleSet[account],
         "ZivoeRewardsVesting::revokeVestingSchedule()
↪  !vestingScheduleSet[account]"
     );
     require(
         vestingScheduleOf[account].revokable,
         "ZivoeRewardsVesting::revokeVestingSchedule()
↪  !vestingScheduleOf[account].revokable"
     );

     uint256 amount = amountWithdrawable(account);
     uint256 vestingAmount = vestingScheduleOf[account].totalVesting;

     vestingTokenAllocated -= amount;

     vestingScheduleOf[account].totalWithdrawn += amount;
     vestingScheduleOf[account].totalVesting =
↪  vestingScheduleOf[account].totalWithdrawn;
     vestingScheduleOf[account].cliff = block.timestamp - 1;
     vestingScheduleOf[account].end = block.timestamp;

     vestingTokenAllocated -= (vestingAmount -
↪  vestingScheduleOf[account].totalWithdrawn);

  -  _totalSupply = _totalSupply.sub(vestingAmount);
  + _totalSupply = _totalSupply.sub(amount);
```

Discussion

pseudonaut

Valid

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> high. Incorrect (reduced) totalSupply will make late users being unable to claim their reward

pseudonaut

Recommended suggestion seems invalid, example (only one vesting schedule):

`1,000` initial vesting schedule `200` claimed already `100` claimable

As a result of the 200 claimed already, totalSupply() should have decreased from `1000 -> 800`

Given current logic, once the vesting schedule is revoked, it would reduce totalSupply() by amount only, from `800 -> 700` ... this leaves surplus of stake ... it should reduce total by the full amount of `800` so that totalSupply() at the end is `0` not `700`

Calculation (as suggested in other PR) should be as follows:

```
_totalSupply = _totalSupply.sub(vestingAmount -
↪   vestingScheduleOf[account].totalWithdrawn + amount);
```

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Zivoe/zivoe-core-foundry/pull/268

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-4: Revoking vesting schedule does not subtract user votes correctly

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/118

Found by

0xAnmol, 0xpiken, 0xvj, 9oelm, Afriaudit, Ironsidesec, JigglypuffAndPikachu, KingNFT, KupiaSec, Ruhum, SilverChariot, Tendency, Tychai0s, Varun_05, amar, denzi_, dimulski, ether_sky, lemonmon, marchev, mt030d, rbserver, saidam017, sakshamguruji, samuraii77

Summary

ZivoeVestingRewards.revokeVestingSchedule() should reduce the voting power of the user with the withdrawable amount plus the revoked amount. However, it reduces it only by the withdrawable amount.

Vulnerability Detail

When called, `revokeVestingSchedule()` fetches the withdrawable amount by the user at that moment.

```
uint256 amount = amountWithdrawable(account);
```

The revoke logic is executed and the user's checkpoint value is decreased by `amount`.

```
_writeCheckpoint(_checkpoints[account], _subtract, amount);
```

The code ignores the amount that's being revoked and the user keeps more voting power than he has to. Imagine the following: `totalVested = 1000` `withdrawable = 0`

If the schedule gets revoked, the user's checkpoint value will not be decreased at all because there is nothing to be withdrawn. The user can later use their voting power to vote on governance proposals.

In fact, `amountWithdrawable(account)` being close to 0 has a very high likelihood because:

- the user can frontrun the transaction and withdraw the tokens they are entitled to
- it's highly likely that a vesting schedule will be removed shortly after creating it.

However, even if `amountWithdrawable()` is not equal to 0, the user would still be left with more voting power.

Impact

Users keep voting power that must have been taken away.

Code Snippet

POC to be run in Test_ZivoeRewardsVesting.sol

```
function test_revoking_leaves_votes() public {
   assert(zvl.try_createVestingSchedule(
        address(vestZVE),
        address(moe),
        0,
        360,
        6000 ether,
        true
    ));
    // Vesting succeeded
    assertEq(vestZVE.balanceOf(address(moe)), 6000 ether);

    hevm.roll(block.number + 1);

    // User votes have increased
    assertEq(vestZVE.getPastVotes(address(moe), block.number - 1), 6000 ether);

    assert(zvl.try_revokeVestingSchedule(address(vestZVE), address(moe)));
    // Revoking succeeded
    assertEq(vestZVE.balanceOf(address(moe)), 0);

    hevm.roll(block.number + 1);
    // User votes have not been decreased at all
    assertEq(vestZVE.getPastVotes(address(moe), block.number - 1), 6000 ether);
}
```

Tool used

Foundry

Recommendation

Subtract the correct amount from the checkpoint's value

```
-        _writeCheckpoint(_checkpoints[account], _subtract, amount);
+       _writeCheckpoint(_checkpoints[account], _subtract, vestingAmount -
↪   vestingScheduleOf[account].totalWithdrawn + amount)
```

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> high, incorrect checkpoints amount subtracted during `revokeVestingSchedule` causes permanent inflated amount of user votes.

pseudonaut

Recommended suggestion seems incorrect

It doesn't address the totalSupplyCheckpoints right above that line (both should be in tandem):

```
_writeCheckpoint(_totalSupplyCheckpoints, _subtract, vestingAmount -
↪   vestingScheduleOf[account].totalWithdrawn + amount);
_writeCheckpoint(_checkpoints[account], _subtract, vestingAmount -
↪   vestingScheduleOf[account].totalWithdrawn + amount);
```

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Zivoe/zivoe-core-foundry/pull/268

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-5: `depositReward()` with zero amount to get reward tokens stuck in `ZivoeRewards` contracts

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/129

Found by

Audinarey, Drynooo, KingNFT, KupiaSec, Maniacs, Quanta, SUPERMAN_I4G, Tricko, beWater0given, dimulski, flacko, joicygiore

Summary

`ZivoeRewards.depositReward()` has no access control, attackers can call it with zero amount of `reward` to get some reward tokens stuck in contract. The locked amount could be significant especially when reward tokens have small decimals such as `USDT/USDC/WBTC`.

Vulnerability Detail

The issue arises due to precision loss on L233 and L237.

```
File: zivoe-core-foundry\src\ZivoeRewards.sol
228:     function depositReward(address _rewardsToken, uint256 reward) external
 ↪  updateReward(address(0)) nonReentrant {
...
232:         if (block.timestamp >= rewardData[_rewardsToken].periodFinish) {
233:             rewardData[_rewardsToken].rewardRate =
 ↪  reward.div(rewardData[_rewardsToken].rewardsDuration);
234:         } else {
...
237:             rewardData[_rewardsToken].rewardRate =
 ↪  reward.add(leftover).div(rewardData[_rewardsToken].rewardsDuration);
238:         }
...
243:     }
```

The following PoC shows two cases that `654 USDC` and `6.6 WBTC` get stuck in `ZivoeRewards(stZVE)` contract after `20` calls on `depositReward()` with zero amount of `reward`.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "../Utility/Utility.sol";
import "forge-std/console2.sol";

contract TestDepositRewardBug is Utility {
    address alice;
    function setUp() public {
        vm.createSelectFork("https://mainnet.gateway.tenderly.co");
        deployCore(false);

        deal(USDC, address(zvl), 1_000e6); // 1,000 USDC
        deal(WBTC, address(zvl), 10e8); // 10 wBTC
        vm.startPrank(address(zvl));
        stZVE.addReward(USDC, 365 days);
        stZVE.addReward(WBTC, 365 days);
        IERC20(USDC).approve(address(stZVE), type(uint256).max);
        stZVE.depositReward(USDC, 1_000e6);
        IERC20(WBTC).approve(address(stZVE), type(uint256).max);
        stZVE.depositReward(WBTC, 10e8);
        vm.stopPrank();


        alice = makeAddr("alice");
        deal(address(ZVE), alice, 1_000e18);
        assertEq(ZVE.balanceOf(alice), 1_000e18);

        vm.startPrank(alice);
        ZVE.approve(address(stZVE), type(uint256).max);
        stZVE.stake(1_000e18);
        vm.stopPrank();

        // alice is the only staker, and is expected to get all rewards
        assertEq(1_000e18, stZVE.totalSupply());
    }

    function testAttackOnUSDCReward() public {
        uint256 timestamp = block.timestamp;
        for (uint256 i; i < 20; ++i) {
            vm.warp(timestamp += 12);
            // deposit nothing
            stZVE.depositReward(USDC, 0);
        }

        vm.warp(timestamp + 365 days + 1); // make sure all reward distributed
        vm.prank(alice);
        stZVE.getRewards();
        uint256 balance = IERC20(USDC).balanceOf(alice);
        assertApproxEqAbs(346e6, balance, 1e6);
        balance = IERC20(USDC).balanceOf(address(stZVE));
        assertApproxEqAbs(654e6, balance, 1e6);
        console2.log("Expected reward: 1,000 USDC, actual reward: 346 USDC,
↪   locked: 654 USDC");
    }

    function testAttackOnWBTCReward() public {
        uint256 timestamp = block.timestamp;
        for (uint256 i; i < 20; ++i) {
            vm.warp(timestamp += 12);
            // deposit nothing
            stZVE.depositReward(WBTC, 0);
        }

        vm.warp(timestamp + 365 days + 1); // make sure all reward distributed
        vm.prank(alice);
        stZVE.getRewards();
        uint256 balance = IERC20(WBTC).balanceOf(alice);
        assertApproxEqAbs(3.4e8, balance, 0.1e8);
        balance = IERC20(WBTC).balanceOf(address(stZVE));
        assertApproxEqAbs(6.6e8, balance, 0.1e8);
        console2.log("Expected reward: 10 WBTC, actual reward: 3.4 WBTC, locked:
↪   6.6 WBTC");
    }
}
```

The test log:

```
2024-03-zivoe\zivoe-core-testing> forge test --mc TestDepositRewardBug -vv
[] Compiling...
No files changed, compilation skipped

Ran 2 tests for src/TESTS_Core/Bug_DepositReward.t.sol:TestDepositRewardBug
[PASS] testAttackOnUSDCReward() (gas: 822088)
Logs:
  Expected reward: 1,000 USDC, actual reward: 346 USDC, locked: 654 USDC

[PASS] testAttackOnWBTCReward() (gas: 788716)
Logs:
  Expected reward: 10 WBTC, actual reward: 3.4 WBTC, locked: 6.6 WBTC

Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 30.37s (5.20s CPU
↪   time)

Ran 1 test suite in 30.41s (30.37s CPU time): 2 tests passed, 0 failed, 0
↪   skipped (2 total tests)
```

Impact

Fund get stuck in contract

Code Snippet

https://github.com/sherlock-audit/2024-03-zivoe/blob/d4111645b19a1ad3ccc899bea073b6f19be04ccd/zivoe-core-foundry/src/ZivoeRewards.sol#L228C14-L228C27

Tool used

Manual Review

Recommendation

Increasing the precision of `rewardRate` by `1e18`.

Discussion

pseudonaut

Valid, considering adding whitelists

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> high, depositRewards leaves some dust amounts which can accumulate and can be substantial for 6-8 decimals tokens

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Zivoe/zivoe-core-foundry/pull/260

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-6: Protocol unable to get extra Rewards in OCY_-Convex_C

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/477

Found by

BoRonGod, cergyk

Summary

Convex would wrap `rewardToken` for pools with IDs 151+, but the counting logic in `OCY_Convex_C.sol` makes it impossible for zivoe to forward yield.

Vulnerability Detail

In `OCY_Convex_C.sol` , A convex pool with id `270` is used:

```
/// @dev Convex information.
address public convexDeposit = 0xF403C135812408BFbE8713b5A23a04b3D48AAE31;
address public convexPoolToken = 0x383E6b4437b59fff47B619CBA855CA29342A8559;
address public convexRewards = 0xc583e81bB36A1F620A804D8AF642B63b0ceEb5c0;

uint256 public convexPoolID = 270;
```

In the following logic, `rewardContract` is defaulted to the address of extraRewards. This assumption is fine for pools with PoolId < 150, but would not work for IDs 151+.

```
/// @notice Claims rewards and forward them to the OCT_YDL.
/// @param extra Flag for claiming extra rewards.
function claimRewards(bool extra) public nonReentrant {
    IBaseRewardPool_OCY_Convex_C(convexRewards).getReward();

    // Native Rewards (CRV, CVX)
    uint256 rewardsCRV = IERC20(CRV).balanceOf(address(this));
    uint256 rewardsCVX = IERC20(CVX).balanceOf(address(this));
    if (rewardsCRV > 0) { IERC20(CRV).safeTransfer(OCT_YDL, rewardsCRV); }
    if (rewardsCVX > 0) { IERC20(CVX).safeTransfer(OCT_YDL, rewardsCVX); }

    // Extra Rewards
    if (extra) {
        uint256 extraRewardsLength =
         ↪  IBaseRewardPool_OCY_Convex_C(convexRewards).extraRewardsLength();
        for (uint256 i = 0; i < extraRewardsLength; i++) {
            address rewardContract =
            ↪  IBaseRewardPool_OCY_Convex_C(convexRewards).extraRewards(i);
            //@Audit incorrect here!
            uint256 rewardAmount = IBaseRewardPool_OCY_Convex_C(rewardContract).
            ↪  rewardToken().balanceOf(address(this));
            if (rewardAmount > 0) { IERC20(rewardContract).safeTransfer(OCT_YDL,
            ↪  rewardAmount); }
        }
    }
}
```

According to convex doc,

> for pools with IDs 151+: VirtualBalanceRewardPool's rewardToken points to a wrapped version of the underlying token. This Token implementation can be found here: https://github.com/convex-eth/platform/blob/main/contracts/contracts/StashTokenWrapper.sol

Just check `convexRewards` of pool 270:

https://etherscan.io/address/0xc583e81bB36A1F620A804D8AF642B63b0ceEb5c0#readContract#F5

For index 0, it returns a VirtualBalanceRewardPool with rewardtoken = 0x85D81Ee851D36423A5784CD3Cb6f1a1193Cb5978. This contract is a `StashTokenWrapper`, which is consistent with what the convex documentation says.

And, when `IBaseRewardPool_OCY_Convex_C(convexRewards).getReward();` is triggered, reward tokens will be unwrapped and send to caller, so rewardAmount will always return 0, means such yield cannot be claimed for zivoe.

```
address rewardContract =
↪  IBaseRewardPool_OCY_Convex_C(convexRewards).extraRewards(i);

//@Audit incorrect here! `rewardToken` is a `StashTokenWrapper`, not the reward
↪  token!

uint256 rewardAmount = IBaseRewardPool_OCY_Convex_C(rewardContract).rewardToken(
↪  ).balanceOf(address(this));
if (rewardAmount > 0) { IERC20(rewardContract).safeTransfer(OCT_YDL,
↪  rewardAmount); }
```

Impact

Users will lose extra rewards from convex pools with IDs 151+.

Code Snippet

https://github.com/sherlock-audit/2024-03-zivoe/blob/main/zivoe-core-foundry/src/lockers/OCY/OCY_Convex_C.sol#L210-L219
https://docs.convexfinance.com/convexfinanceintegration/baserewardpool

Tool used

Manual Review

Recommendation

Change the logic above to:

```
uint256 rewardAmount = IBaseRewardPool_OCY_Convex_C(rewardContract).rewardToken(
    ).token().balanceOf(address(this));
if (rewardAmount > 0) { IERC20(rewardContract).safeTransfer(OCT_YDL,
    rewardAmount); }
```

Discussion

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> high, loss of additional reward from convex. Since it accumulates over 30-days period, the loss will be significant. Both `balanceOf` and `safeTransfer` are incorrect - they should be called on `rewardToken().token()`. Different dups of this mention either `balanceOf` or `safeTransfer`, #294 mentions both (but this one is more detailed), but I consider them to be the same root cause, so all are dups.

RealLTDingZhen

escalate

I would escalate this to a solo issue.

Lets compare this one with #479 (and all current dups):

#477 only works with convex pool id 151+ , so only `OCY_Convex_C` is affected. #479 works with ALL three `OCY_Convex`.

#477 points out a incorrect external call with convex's `StashTokenWrapper`. #479 points out a incorrect external call with convex's `VirtualBalanceRewardPool`.

#477 and #479 have different fixes. The fix for one cannot fix the other.

#477 and #479 have different impacts.

- If we fix 479 , In 477 , claimRewards will still not forward any extrareward to YDL.
- If we fix 477 , In 479 , claimRewards will still always revert.

#477 and #479 happens on different lines in `OCY_Convex_C`, with different external calls.

```
//#477 root cause: convex would wrap rewards into a StashTokenWrapper
uint256 rewardAmount = IBaseRewardPool_OCY_Convex_C(rewardContract).rewardToken(
↪  ).balanceOf(address(this));

//#479 and all dups root cause: convex rewardContract is a
↪  VirtualBalanceRewardPool
if (rewardAmount > 0) { IERC20(rewardContract).safeTransfer(OCT_YDL,
↪  rewardAmount); }
```

So, why should we duplicate two valid issues **when they have different root causes, different fixes, different impacts, and the root cause of both issues do not have any code reuse?**

I believe Lead judge's statement

> The core reason is still incorrect address for these functions.

should not be the reason to duplicate this issue. These two different incorrect addresses are obtained through different improper external calls towards different external contracts. I don't understand why this issue was considered as dup of #479.

sherlock-admin3

> escalate
>
> I would escalate this to a solo issue.
>
> Lets compare this one with #479 (and all current dups):
>
> #477 only works with convex pool id 151+ , so only `OCY_Convex_C` is affected. #479 works with ALL three `OCY_Convex`.
>
> #477 points out a incorrect external call with convex's `StashTokenWrapper`. #479 points out a incorrect external call with convex's `VirtualBalanceRewardPool`.
>
> #477 and #479 have different fixes. The fix for one cannot fix the other.
>
> #477 and #479 have different impacts.

- If we fix 479 , In 477 , claimRewards will still not forward any extrareward to YDL.
- If we fix 477 , In 479 , claimRewards will still always revert.

#477 and #479 happens on different lines in `OCY_Convex_C`, with different external calls.

```
//#477 root cause: convex would wrap rewards into a StashTokenWrapper
uint256 rewardAmount = IBaseRewardPool_OCY_Convex_C(rewardContract).rewardT
↪  oken().balanceOf(address(this));

//#479 and all dups root cause: convex rewardContract is a
↪  VirtualBalanceRewardPool
if (rewardAmount > 0) { IERC20(rewardContract).safeTransfer(OCT_YDL,
↪  rewardAmount); }
```

So, why should we duplicate two valid issues **when they have different root causes, different fixes, different impacts, and the root cause of both issues do not have any code reuse?**

I believe Lead judge's statement

> The core reason is still incorrect address for these functions.

should not be the reason to duplicate this issue. These two different incorrect addresses are obtained through different improper external calls towards different external contracts. I don't understand why this issue was considered as dup of #479.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

panprog

The core reason is still incorrect address for these functions. The fix is different for different contracts, some issues describe it in more details, some in less. Since `balanceOf` and `transfer` have to happen on the same address, I don't think separate issues for `balanceOf` or `transfer` warrant it being a separate group. The same with _A or _C - it's basically the same, even if the fix in each of them is different. I agree that your issue describes everything much better and the recommendation is more correct compared to dups, but still the core reason is the same. If these nuances warrant a separate group - this will have to be decided by Sherlock. I keep my decision here.

As for the severity: DAO can't pull these tokens, because the only function to pull tokens is overriden for this locker and only allows to pull convex pool token and nothing else.

pseudonaut

I'm confused by the final recommendation for fixes here, can someone provide detailed explanation of fixes for appropriate contracts based on all findings?

panprog

> I'm confused by the final recommendation for fixes here, can someone provide detailed explanation of fixes for appropriate contracts based on all findings?

@pseudonaut , the address of the extra reward token is:

- for convex pools < 151 (OCY_Convex_A): `address = IBaseRewardPool_OCY_Convex_A(rewardContract).rewardToken()`
- for convex pools >= 151 (OCY_Convex_C): `address = IBaseRewardPool_OCY_Convex_A(rewardContract).rewardToken().token()` (you'll need to add token() in the reward token interface)

Both `balanceOf` and `safeTransfer` should be done on this address. Something like

```
if (extra) {
    uint256 extraRewardsLength =
↪   IBaseRewardPool_OCY_Convex_A(convexRewards).extraRewardsLength();
    for (uint256 i = 0; i < extraRewardsLength; i++) {
        address rewardContract =
↪   IBaseRewardPool_OCY_Convex_A(convexRewards).extraRewards(i);
        IERC20 rewardToken = convexPoolID < 151 ?
↪   IBaseRewardPool_OCY_Convex_A(rewardContract).rewardToken() :
↪   IBaseRewardPool_OCY_Convex_A(rewardContract).rewardToken().token();
        uint256 rewardAmount = rewardToken.balanceOf(
            address(this)
        );
        if (rewardAmount > 0) { IERC20(rewardToken).safeTransfer(OCT_YDL,
↪   rewardAmount); }
    }
}
```

WangSecurity

Firstly, I agree there are two issues:

1. `rewardAmount` is called on the `rewardToken`, when it should be called on `rewardToken().token`

2. `safeTransfer` is called on `rewardContract` when it should be called on `rewardToken`.

Fixing one, doesn't fix the other. I agree that general core issue is in fact using the wrong address, but the fixes and impacts are different (for 1 is not paying out the extra rewards and they're kept in the contract, for 2 is the revert of the call). Hence, I believe they should be treated separately. There is only 1 report that describes both of these vulnerabilities, which is #294, therefore, I believe it would be fair to make the following issue families:

1. Getting `rewardAmount` on `rewardToken`:

- #294
- #477 (this one) - best

2. Calling `safeTransfer` on `rewardContract`:

- #161
- #222
- #277
- #391
- #399
- #444
- #447
- #479 - best

The reason why #294 is in the 1st family is because it mentions both issues, but since one report cannot get 2 rewards, it'll be fair to include it in the family with less reports.

Lastly, I believe in both situations there are no extensive external limitations leading to a loss of funds. Please correct me if any assumption above is wrong.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Zivoe/zivoe-core-foundry/pull/273

panprog

@WangSecurity Agree with your decision. Keeping #294 a dup of this also looks correct.

Evert0x

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- RealLTDingZhen: accepted

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-7: cannot forward extra rewards from both OCY_-Convex to OCT_YDL.

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/479

Found by

0xpiken, AllTooWell, BoRonGod, Drynooo, Ironsidesec, den_sosnovskyi, ether_sky

Summary

Convex specifies `rewardContract` to be a `VirtualBalanceRewardPool`, but all three OCY_Convex uses it as a ERC20 token, which make it impossible to claim extra rewards and forward them to the OCT_YDL.

Vulnerability Detail

According to Convex doc:

> The BaseRewardPool has an array of child reward contracts called extraRewards. You can query the number of extra rewards via baseRewardPool.extraRewardsLength(). This array holds a list of VirtualBalanceRewardPool contracts which are similar in nature to the base reward contract but without actual control of staked tokens.
>
> This means that if a pool has CRV rewards as well as SNX rewards, the pool's main reward contract(BaseRewardPool) will distribute the CRV and the child contract(VirtualBalanceRewardPool) will distribute the SNX.

But, in current implementation: (take OCY_Convex_A for example)

```
// Extra Rewards
if (extra) {
    uint256 extraRewardsLength =
    ↪   IBaseRewardPool_OCY_Convex_A(convexRewards).extraRewardsLength();
    for (uint256 i = 0; i < extraRewardsLength; i++) {
        address rewardContract =
        ↪   IBaseRewardPool_OCY_Convex_A(convexRewards).extraRewards(i);
        uint256 rewardAmount =
        ↪   IBaseRewardPool_OCY_Convex_A(rewardContract).rewardToken().balanceOf(
            address(this)
        );
        //@Audit rewardContract is a VirtualBalanceRewardPool
        if (rewardAmount > 0) { IERC20(rewardContract).safeTransfer(OCT_YDL,
        ↪   rewardAmount); }
    }
}
```

Tokens cannot be sent to YDL.

Impact

Current OCY_Convex_A, OCY_Convex_B and OCY_Convex_C cannot forward extraRewards to YDL.

Code Snippet

https://github.com/sherlock-audit/2024-03-zivoe/blob/main/zivoe-core-foundry/src/lockers/OCY/OCY_Convex_A.sol#L263 https://docs.convexfinance.com/convexfinanceintegration/baserewardpool#extra-rewards

Tool used

Manual Review

Recommendation

Use the real token address for token transfer.

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> high, dup of #477, loss of additional reward from convex. Since it accumulates over 30-days period, the loss will be significant. Both `balanceOf` and `safeTransfer` are incorrect - they should be called on `rewardToken().token()`. Different dups of this mention either `balanceOf` or `safeTransfer`, but not both, but I consider them to be the same root cause, so all are dups.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Zivoe/zivoe-core-foundry/pull/273

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-1: ZivoeYDL::distributeYield() will revert if protocolRecipients recipients length is smaller than residualRecipients

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/15

Found by

BiasedMerc

Summary

`ZivoeYDL::distributeYield` incorrectly accesses `_protocol[i]` instead of `_residual[i]`, and there are no checks anywhere within the code that ensures that both arrays are of the same length. Meaning that it is likely that this will cause `distributeYield()` to revert in the case where `protocolRecipients` length is less than `residualRecipients`.

Vulnerability Detail

`ZivoeYDL::updateRecipients` allows the timelock contract to update the `protocolRecipients` or `residualRecipients` variables:

ZivoeYDL::updateRecipients

```
    function updateRecipients(address[] memory recipients, uint256[] memory
 ↪  proportions, bool protocol) external {
        require(_msgSender() == IZivoeGlobals_YDL(GBL).TLC(),
 ↪  "ZivoeYDL::updateRecipients() _msgSender() != TLC()");
        require(
            recipients.length == proportions.length && recipients.length > 0,
            "ZivoeYDL::updateRecipients() recipients.length !=
 ↪  proportions.length || recipients.length == 0"
        );
        require(unlocked, "ZivoeYDL::updateRecipients() !unlocked");
... SKIP!...
        if (protocol) {
            emit UpdatedProtocolRecipients(recipients, proportions);
            protocolRecipients = Recipients(recipients, proportions);
        }
        else {
            emit UpdatedResidualRecipients(recipients, proportions);
            residualRecipients = Recipients(recipients, proportions);
        }
    }
```

It's important to note this function only sets one of those 2 variables at a time, and the length checks on `recipients` and `proportions` is to ensure that the `Recipients()` struct inputs have the same length, NOT to ensure that `protocolRecipients` and `residualRecipients` are the same length.

`ZivoeYDL::distributeYield` calculates protocol earnings and distributes the yield to both the `protocolRecipients` and `residualRecipients`. It does so by looping through the recipients and the earnings:

ZivoeYDL::distributeYield

```
    function distributeYield() external nonReentrant {
        require(unlocked, "ZivoeYDL::distributeYield() !unlocked");
        require(
            block.timestamp >= lastDistribution + daysBetweenDistributions *
↪   86400,
            "ZivoeYDL::distributeYield() block.timestamp < lastDistribution +
↪   daysBetweenDistributions * 86400"
        );

        // Calculate protocol earnings.
        uint256 earnings = IERC20(distributedAsset).balanceOf(address(this));
        uint256 protocolEarnings = protocolEarningsRateBIPS * earnings / BIPS;
        uint256 postFeeYield = earnings.floorSub(protocolEarnings);

        // Update timeline.
        distributionCounter += 1;
        lastDistribution = block.timestamp;

        // Calculate yield distribution (trancheuse = "slicer" in French).
        (
            uint256[] memory _protocol, uint256 _seniorTranche, uint256
↪   _juniorTranche, uint256[] memory _residual
        ) = earningsTrancheuse(protocolEarnings, postFeeYield);

... SKIP!...

        // Distribute protocol earnings.
        for (uint256 i = 0; i < protocolRecipients.recipients.length; i++) {
            address _recipient = protocolRecipients.recipients[i];
            if (_recipient == IZivoeGlobals_YDL(GBL).stSTT() ||_recipient ==
↪   IZivoeGlobals_YDL(GBL).stJTT()) {
                IERC20(distributedAsset).safeIncreaseAllowance(_recipient,
↪   _protocol[i]);
```

```
                IZivoeRewards_YDL(_recipient).depositReward(distributedAsset,
↪   _protocol[i]);
                emit YieldDistributedSingle(distributedAsset, _recipient,
↪   _protocol[i]);
            }

... SKIP!...

        // Distribute residual earnings.
        for (uint256 i = 0; i < residualRecipients.recipients.length; i++) {
            if (_residual[i] > 0) {
                address _recipient = residualRecipients.recipients[i];
                if (_recipient == IZivoeGlobals_YDL(GBL).stSTT() ||_recipient ==
↪   IZivoeGlobals_YDL(GBL).stJTT()) {
                    IERC20(distributedAsset).safeIncreaseAllowance(_recipient,
↪   _residual[i]);

↪   IZivoeRewards_YDL(_recipient).depositReward(distributedAsset, _residual[i]);
                    emit YieldDistributedSingle(distributedAsset, _recipient,
↪   _protocol[i]);
                }
```

The function loops over `protocolRecipients` and `residualRecipients` seperately and within each loop indexed positions are accessed in `_protocol[]` and `_residual[]`, these 2 arrays are initated as follows:

ZivoeYDL::earningsTrancheuse

```
function earningsTrancheuse(uint256 yP, uint256 yD) public view returns (
    uint256[] memory protocol, uint256 senior, uint256 junior, uint256[] memory
↪   residual
) {
    protocol = new uint256[](protocolRecipients.recipients.length);
    residual = new uint256[](residualRecipients.recipients.length);
```

Their lengths will depend on the lengths of `protocolRecipients` and `residualRecipients` respectively, meaning their lengths can differ.

Finally, the 2nd for loop in `distributeYield` itterates over `residualRecipients` and incorrectly emits the `YieldDistributedSingle` event by accessing `_protocol[i]`, whilst it should be using `_residual[i]`. This will cause the function to revert due to an out of bound index access if:

`residualRecipients.recipients.length > protocolRecipients.recipients.length`
AND atleast one of the recipients in `residualRecipients.recipients` is:
`IZivoeGlobals_YDL(GBL).stSTT() || IZivoeGlobals_YDL(GBL).stJTT()`

which are the reward contracts: ZivoeYDL.sol#L20-L24

```
/// @notice Returns the address of the ZivoeRewards ($zSTT) contract.
function stSTT() external view returns (address);

/// @notice Returns the address of the ZivoeRewards ($zJTT) contract.
function stJTT() external view returns (address);
```

meaning yield will not be able to be successfully distributed and the function will always revert as long as this condition is met.

Impact

Core functionality of `ZivoeYDL` will be unaccessible due to the function always reverting as long as the above condition is met, and this condition can easily be met through normal protocol use.

This can be fixed by ensuring that both array are of the same length, however this means that appropriate fees cannot be distributed to the correct parties.

Code Snippet

ZivoeYDL.sol#L392-L416 ZivoeYDL.sol#L213-L286 ZivoeYDL.sol#L447-L451 ZivoeYDL.sol#L20-L24

Tool used

Manual Review

Recommendation

Change the incorrect emit to access from `_residual[i]` rather than `_protocol[i]`: ZivoeYDL.sol#L286

```
- emit YieldDistributedSingle(distributedAsset, _recipient, _protocol[i]);
+ emit YieldDistributedSingle(distributedAsset, _recipient, _residual[i]);
```

Discussion

pseudonaut

Valid

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> medium, yield distribution always reverts in some cases if protocols recepients < residual recepients

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Zivoe/zivoe-core-foundry/pull/253

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-2: OCL_ZVE::pushToLockerMulti() will revert due to incorrect assert() statements when interacting with UniswapV2

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/18

Found by

0rpse, 0x73696d616f, 0xAnmol, 0xboriskataa, 0xpiken, 0xvj, AMOW, Afriaudit, AllTooWell, BiasedMerc, BoRonGod, Drynooo, FastTiger, Ironsidesec, JigglypuffAndPikachu, KupiaSec, Maniacs, Quanta, Ruhum, SilverChariot, Tendency, blackhole, blockchain555, cergyk, dany.armstrong90, den_sosnovskyi, ether_sky, flacko, jasonxiale, krikolkk, lemonmon, mt030d, recursiveEth, saidam017, sl1

Summary

`OCL_ZVE::pushToLockerMulti()` verifies that the allowances for both tokens is 0 after providing liquidity to UniswapV2 or Sushi routers, however there is a high likelihood that one allowance will not be 0, due to setting a 90% minimum liquidity provided value. Therefore, the function will revert most of the time breaking core functionality of the locker, making the contract useless.

Vulnerability Detail

The DAO can add liquidity to UniswapV2 or Sushi through `OCL_ZVE::pushToLockerMulti()` function, where `addLiquidity` is called on `router`:

OCL_ZVE.sol#L198C78-L198

```
IRouter_OCL_ZVE(router).addLiquidity(
```

OCL_ZVE.sol#L90

```
address public immutable router;            /// @dev Address for the Router
↪  (Uniswap v2 or Sushi).
```

The router is intended to be Uniswap v2 or Sushi (Sushi router uses the same code as Uniswap v2 0xd9e1ce17f2641f24ae83637ab66a2cca9c378b9f).

UniswapV2Router02::addLiquidity

```
function addLiquidity(
    address tokenA,
    address tokenB,
    uint amountADesired,
    uint amountBDesired,
    uint amountAMin,
    uint amountBMin,
    address to,
    uint deadline
) external virtual override ensure(deadline) returns (uint amountA, uint
↪   amountB, uint liquidity) {
    (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,
↪   amountBDesired, amountAMin, amountBMin);
    address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);
    TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
    TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
    liquidity = IUniswapV2Pair(pair).mint(to);
}
```

When calling the function 4 variables relevant to this issue are passed: `amountADesired` and `amountBDesired` are the ideal amount of tokens we want to deposit, whilst `amountAMin` and `amountBMin` are the minimum amounts of tokens we want to deposit. Meaning the true amount that will deposit be deposited for each token will be inbetween those 2 values, e.g: `amountAMin <= amountA <= amountADesired`. Where `amountA` is how much of `tokenA` will be transfered.

The transfered amount are `amountA` and `amountB` which are calculated as follows: UniswapV2Router02::_addLiquidity

```
function _addLiquidity(
    address tokenA,
    address tokenB,
    uint amountADesired,
    uint amountBDesired,
    uint amountAMin,
    uint amountBMin
) internal virtual returns (uint amountA, uint amountB) {
    // create the pair if it doesn't exist yet
    if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
        IUniswapV2Factory(factory).createPair(tokenA, tokenB);
    }
    (uint reserveA, uint reserveB) = UniswapV2Library.getReserves(factory,
↪   tokenA, tokenB);
    if (reserveA == 0 && reserveB == 0) {
        (amountA, amountB) = (amountADesired, amountBDesired);
    } else {
        uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA,
↪   reserveB);
```

```
        if (amountBOptimal <= amountBDesired) {
            require(amountBOptimal >= amountBMin, 'UniswapV2Router:
↪   INSUFFICIENT_B_AMOUNT');
            (amountA, amountB) = (amountADesired, amountBOptimal);
        } else {
            uint amountAOptimal = UniswapV2Library.quote(amountBDesired,
↪   reserveB, reserveA);
            assert(amountAOptimal <= amountADesired);
            require(amountAOptimal >= amountAMin, 'UniswapV2Router:
↪   INSUFFICIENT_A_AMOUNT');
            (amountA, amountB) = (amountAOptimal, amountBDesired);
        }
    }
}
```

`UniswapV2Router02::_addLiquidity` receives a quote for how much of each token can be added and validates that the values fall within the `amountAMin` and `amountADesired` range. Unless the exactly correct amounts are passed as `amountADesired` and `amountBDesired` then the amount of one of the two tokens will be less than the desired amount.

Now lets look at how `OCL_ZVE` interacts with the Uniswapv2 router:

OCL_ZVE::addLiquidity

```
// Router addLiquidity() endpoint.
uint balPairAsset = IERC20(pairAsset).balanceOf(address(this));
uint balZVE = IERC20(ZVE).balanceOf(address(this));
IERC20(pairAsset).safeIncreaseAllowance(router, balPairAsset);
IERC20(ZVE).safeIncreaseAllowance(router, balZVE);

// Prevent volatility of greater than 10% in pool relative to amounts present.
(uint256 depositedPairAsset, uint256 depositedZVE, uint256 minted) =
↪   IRouter_OCL_ZVE(router).addLiquidity(
    pairAsset,
    ZVE,
    balPairAsset,
    balZVE,
    (balPairAsset * 9) / 10,
    (balZVE * 9) / 10,
    address(this), block.timestamp + 14 days
);
emit LiquidityTokensMinted(minted, depositedZVE, depositedPairAsset);
assert(IERC20(pairAsset).allowance(address(this), router) == 0);
assert(IERC20(ZVE).allowance(address(this), router) == 0);
```

The function first increases the allowances for both tokens to `balPairAsset` and `balZVE` respectively.

When calling the router, `balPairAsset` and `valZVE` are provided as the desired amount of liquidity to add, however `(balPairAsset * 9) / 10` and `(balZVE * 9) / 10` are also passed as minimums for how much liquidity we want to add.

As the final transfered value will be between: `(balPairAsset * 9) / 10 <= x <= balPairAsset` therefore the allowance after providing liquidity will be: `0 <= IERC20(pairAsset).allowance(address(this), router) <= balPairAsset - (balPairAsset * 9) / 10` however the function expects the allowance to be 0 for both tokens after providing liquidity. The same applies to the `ZVE` allowance.

This means that in most cases one of the assert statements will not be met, leading to the add liquidity call to revert. This is unintended behaviour, as the function passed a `90%` minimum amount, however the allowance asserts do not take this into consideration.

Impact

Calls to `OCL_ZVE::pushToLockerMulti()` will revert a majority of the time, causing core functionality of providing liquidity through the locker to be broken.

Code Snippet

OCL_ZVE.sol#L198C78-L198 UniswapV2Router02.sol#L61-L76 UniswapV2Router02.sol#L33-L60 OCL_ZVE.sol#L191-L209

Tool used

Manual Review

Recommendation

The project wants to clear allowances after all transfers, therefore set the router allowance to 0 after providing liquidity using the returned value from the router:

```
  (uint256 depositedPairAsset, uint256 depositedZVE, uint256 minted) =
↪   IRouter_OCL_ZVE(router).addLiquidity(
      pairAsset,
      ZVE,
      balPairAsset,
      balZVE,
      (balPairAsset * 9) / 10,
      (balZVE * 9) / 10,
      address(this), block.timestamp + 14 days
  );
  emit LiquidityTokensMinted(minted, depositedZVE, depositedPairAsset);
- assert(IERC20(pairAsset).allowance(address(this), router) == 0);
- assert(IERC20(ZVE).allowance(address(this), router) == 0);
+ uint256 pairAssetAllowanceLeft = balPairAsset - depositedPairAsset;
+ if (pairAssetAllowanceLeft > 0) {
+     IERC20(pairAsset).safeDecreaseAllowance(router, pairAssetAllowanceLeft);
+ }
+ uint256 zveAllowanceLeft = balZVE - depositedZVE;
+ if (zveAllowanceLeft > 0) {
+     IERC20(ZVE).safeDecreaseAllowance(router, zveAllowanceLeft);
+ }
```

This will remove the left over allowance after providing liquidity, ensuring the allowance is 0.

Discussion

pseudonaut

Valid

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> medium. While amounts provided are from the trusted admin, who is supposed to provide exact amounts to add liquidity in correct ratio of token amounts, the fact that live balance of the contract is used makes it possible for the balance to change from the time admin provides the amounts, making the transaction revert. Additionally, current contract balances might be in so high disproportion, that admin will simply not have enough funds to create correct ratio to add liquidity.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Zivoe/zivoe-core-foundry/pull/264

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-3: distributeYield() calls earningsTrancheuse() with outdated emaSTT & emaJTT while calculating senior & junior tranche yield distributions

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/54

Found by

Ironsidesec, Nihavent, t0x1c

Summary

The distributeYield() function internally calls `earningsTrancheuse()` on L232 which calculates & returns the `_seniorTranche` & `_juniorTranche` yield distribution. However, this call results in `earningsTrancheuse()` using outdated values of `emaSTT` & `emaJTT` on L461-L462 as these variables are updated only on L238-L239 *after* the call to `earningsTrancheuse()` is concluded.

Code Snippet

```
File: src/ZivoeYDL.sol

213:              function distributeYield() external nonReentrant {
214:                  require(unlocked, "ZivoeYDL::distributeYield() !unlocked");
215:                  require(
216:                      block.timestamp >= lastDistribution +
↪   daysBetweenDistributions * 86400,
217:                      "ZivoeYDL::distributeYield() block.timestamp <
↪   lastDistribution + daysBetweenDistributions * 86400"
218:                  );
219:
220:                  // Calculate protocol earnings.
221:                  uint256 earnings =
↪   IERC20(distributedAsset).balanceOf(address(this));
222:                  uint256 protocolEarnings = protocolEarningsRateBIPS *
↪   earnings / BIPS;
223:                  uint256 postFeeYield = earnings.floorSub(protocolEarnings);
224:
225:                  // Update timeline.
226:                  distributionCounter += 1;
227:                  lastDistribution = block.timestamp;
228:
229:                  // Calculate yield distribution (trancheuse = "slicer" in
↪   French).
```

```
230:                    (
231:                        uint256[] memory _protocol, uint256 _seniorTranche,
 ↪  uint256 _juniorTranche, uint256[] memory _residual
232: @--->                ) = earningsTrancheuse(protocolEarnings, postFeeYield);
233:
234:                    emit YieldDistributed(_protocol, _seniorTranche,
 ↪  _juniorTranche, _residual);
235:
236:                    // Update ema-based supply values.
237:                    (uint256 aSTT, uint256 aJTT) =
 ↪  IZivoeGlobals_YDL(GBL).adjustedSupplies();
238: @--->              emaSTT = MATH.ema(emaSTT, aSTT,
 ↪  retrospectiveDistributions.min(distributionCounter));
239: @--->              emaJTT = MATH.ema(emaJTT, aJTT,
 ↪  retrospectiveDistributions.min(distributionCounter));
                    ...
                    ...
```

and

```
File: src/ZivoeYDL.sol

447:                function earningsTrancheuse(uint256 yP, uint256 yD) public
 ↪  view returns (
448:                    uint256[] memory protocol, uint256 senior, uint256 junior,
 ↪  uint256[] memory residual
449:                ) {
450:                    protocol = new
 ↪  uint256[](protocolRecipients.recipients.length);
451:                    residual = new
 ↪  uint256[](residualRecipients.recipients.length);
452:
453:                    // Accounting for protocol earnings.
454:                    for (uint256 i = 0; i <
 ↪  protocolRecipients.recipients.length; i++) {
455:                        protocol[i] = protocolRecipients.proportion[i] * yP /
 ↪  BIPS;
456:                    }
457:
458:                    // Accounting for senior and junior earnings.
459:                    uint256 _seniorProportion = MATH.seniorProportion(
460:                        IZivoeGlobals_YDL(GBL).standardize(yD,
 ↪  distributedAsset),
461: @--->                  MATH.yieldTarget(emaSTT, emaJTT, targetAPYBIPS,
 ↪  targetRatioBIPS, daysBetweenDistributions),
462: @--->                  emaSTT, emaJTT, targetAPYBIPS, targetRatioBIPS,
 ↪  daysBetweenDistributions
```

```
                        );
                        senior = (yD * _seniorProportion) / RAY;
                        junior = (yD * MATH.juniorProportion(emaSTT, emaJTT,
 ↪  _seniorProportion, targetRatioBIPS)) / RAY;

                        // Handle accounting for residual earnings.
                        yD = yD.floorSub(senior + junior);
                        for (uint256 i = 0; i <
 ↪  residualRecipients.recipients.length; i++) {
                            residual[i] = residualRecipients.proportion[i] * yD /
 ↪  BIPS;
                        }
                    }
```

Vulnerability Detail

As the docs explain, the EMA plays a significant role in yield distribution calculations.

> Returns are calculated using the adjusted supply of tranche tokens, with an Exponential Moving Average (EMA) playing a significant role.
>
> The EMA smoothens the change in the supply of tranche tokens over a look-back period of 2.5 months.

The EMA is used to calculate the:

- yield target via a call to MATH.yieldTarget() which expects "*ema-based supply of zSTT & zJTT*" as its params.
- senior tranche yield proportion via a call to MATH.seniorProportion() which again expects ema-based `yT, eSTT & eJTT`.

Both the above mentioned calls happen inside `earningsTrancheuse()` here. The issue is that these global variables `emaSTT and emaJTT` are still outdated and correspond to the ones belonging to the `lastDistribution` timestamp instead of current updated ones. These values are updated only after the call to `earningsTrancheuse()` has concluded here.

Impact

Imagine that in the last 30 days, JTT supply has reduced due to defaulted loans. As a result, the target yield yT_{real} would come down too (i.e. $yT_{real} < yT$ where yT is the protocol calculated incorrect target yield) and the junior tranche ditributable yield will be smaller than before. However if there is excess yield, then the junior tranche would receive more than their fair share since last 30 days have not been considered by the protocol calculations. If the quantum of defaulted loans is high (and hence yT_{real} has dipped significantly), the impact is not only limited to the junior tranche, but then also effects the senior tranche.

On the flip side an increase in the adjusted supplies of STT and JTT in the last 30 days will have a impact on the smoothened emaSTT and emaJTT, making the values go higher. As a result, target yield yT_{real} would go up too. Thus, it could happen that `yD` is less than yT_{real} but the protocol is oblivious to that due to usage of outdated values. This would result in the protcol using seniorProportionBase() to calculate the senior's yield instead of seniorProportionShortfall().

Finally, since at each of these function calls an outdated eSTT is passed as a param, the value returned would be more/less than what it should be, thus causing gain/loss of yield for both the tranches (juniorProportion() calculation has the same problem).

Note: For the very first distribution i.e. when `distributionCounter = 1`, the values used are from 60 days ago instead of 30 days as can be seen inside unlock(), further increasing the margin of error. The 60 day gap has initially been provided by the protocol to allow for more time to bring in initial yield.

Tool used

Manual Review

Recommendation

Update the ema values ***before*** calling `earningsTrancheuse()`:

```
+       // Update ema-based supply values.
+       (uint256 aSTT, uint256 aJTT) = IZivoeGlobals_YDL(GBL).adjustedSupplies();
+       emaSTT = MATH.ema(emaSTT, aSTT,
↪   retrospectiveDistributions.min(distributionCounter));
+       emaJTT = MATH.ema(emaJTT, aJTT,
↪   retrospectiveDistributions.min(distributionCounter));

        // Calculate yield distribution (trancheuse = "slicer" in French).
        (
            uint256[] memory _protocol, uint256 _seniorTranche, uint256
↪   _juniorTranche, uint256[] memory _residual
        ) = earningsTrancheuse(protocolEarnings, postFeeYield);

        emit YieldDistributed(_protocol, _seniorTranche, _juniorTranche,
↪   _residual);

-       // Update ema-based supply values.
-       (uint256 aSTT, uint256 aJTT) = IZivoeGlobals_YDL(GBL).adjustedSupplies();
-        emaSTT = MATH.ema(emaSTT, aSTT,
 ↪   retrospectiveDistributions.min(distributionCounter));
-        emaJTT = MATH.ema(emaJTT, aJTT,
 ↪   retrospectiveDistributions.min(distributionCounter));
```

Discussion

pseudonaut

Not an issue, the yield that accumulated and is for distribution was generated over the last 30 days and thus should match the initial fix-point of the prior 30 day EMA calculation. There's dilution that occurs with this methodology that we don't want to introduce into our overall accounting system

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> medium, distributeYield uses outdated ema values, updating them after all calculations. However, the impact is limited, because the expected EMA averaging period is 2 months and it's unlikely that distributeYield is called so rarely that the impact of outdated ema values causes any real issues.

panprog

After considering developer's response and intentions, changing this to invalid.

According to developers response, this is intentional, because using current EMA (which includes current totalSupply at the time of distribution, which might be manipulated) can introduce some attack vectors from the current totalSupply, so current distribution is done over EMA from the previous period.

panprog

Sponsor response:

> we did a run-through on accounting scenarios It is indeed underpaying the depositors By what we think is fair within the system And manipulation isn't possible because they'd have to deposit and lock up tokens so there's no flash-loan etc.

Agree with sponsor, changing back to Medium

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Zivoe/zivoe-core-foundry/pull/263

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-4: Forwarding yield in `OCL_ZVE` is possible a lot more often than the enforced 30 days

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/55

Found by

0xpiken, Krace, cergyk, lemonmon, samuraii77

Summary

Calling `OCL_ZVE::forwardYield()` is supposed to only be possible every 30 days however in reality, it is possible to call the function a lot more often.

Vulnerability Detail

The `require` statement in `OCL_ZVE::forwardYield()` enforces that the function should only be callable when `block.timestamp` is of larger value than the value of `nextYieldDistribution`

```
function forwardYield() external {
    if (IZivoeGlobals_OCL_ZVE(GBL).isKeeper(_msgSender())) {
        require(
            block.timestamp > nextYieldDistribution - 12 hours,
            "OCL_ZVE::forwardYield() block.timestamp <= nextYieldDistribution -
↪   12 hours"
        );
    }
    else {
        require(
            block.timestamp > nextYieldDistribution,
            "OCL_ZVE::forwardYield() block.timestamp <= nextYieldDistribution"
        );
    }

    (uint256 amount, uint256 lp) = fetchBasis();
    if (amount > basis) { _forwardYield(amount, lp); }
    (basis,) = fetchBasis();
    nextYieldDistribution += 30 days;
}
```

Then, at the end of the function, `nextYieldDistribution` is incremented by 30 days making the function supposedly not callable for another 30 days. However, there is a vulnerability that allows `OCL_ZVE::forwardYield()` to be called a lot more times.

Upon calling the `OCL_ZVE::pushToLockerMulti()`, the `nextYieldDistribution` gets set to `block.timestamp + 30 days` if its value is 0.

```
if (nextYieldDistribution == 0) { nextYieldDistribution = block.timestamp + 30
↪  days; }
```

If this is the way `nextYieldDistribution` gets its first value, then there will not be an issue. However, if `OCL_ZVE::forwardYield()` is called beforehand, then `nextYieldDistribution` will be set to `0 += 30 days` which equals `30 days` making its value a lot less than the value of `block.timestamp` resulting in people being able to call `OCL_ZVE::forwardYield()` at will all the way until `nextYieldDistribution` gets incremented all the way to `block.timestamp`.

Calling `OCL_ZVE::forwardYield()` before any other function is possible and requires just 1 simple circumstance to be a fact.

Imagine the following scenario:

1. `OCL_ZVE::forwardYield()` is called
2. The `if` statement passes as `block.timestamp` is larger than `nextYieldDistribution`, the value of which is the default value of 0
3. `OCL_ZVE::fetchBasis()` gets called

```
function fetchBasis() public view returns (uint256 amount, uint256 lp) {
        address pool = IFactory_OCL_ZVE(factory).getPair(pairAsset,
↪   IZivoeGlobals_OCL_ZVE(GBL).ZVE());
        uint256 pairAssetBalance = IERC20(pairAsset).balanceOf(pool);
        uint256 poolTotalSupply = IERC20(pool).totalSupply();
        lp = IERC20(pool).balanceOf(address(this));
        amount = lp * pairAssetBalance / poolTotalSupply;
    }
```

4. As long as there is a pool setup for `pairAsset` and `ZVE`, the vulnerability will take place
5. Pool gets the value of the pool address
6. The only thing which has to pass here is `poolTotalSupply` not being 0 as division by 0 is not possible
7. That successfully passes as there is already a setup pool for `pairAsset` and `ZVE`
8. Then, we get back to the `OCL_ZVE::forwardYield()` function
9. `if(amount > basis)` does not pass as both values are 0
10. On the last line, `nextYieldDistribution` gets set to `30 days` making the `OCL_ZVE::forwardYield()` function callable again and again

Impact

`OCL_ZVE::forwardYield()` is callable over and over again even though it is only supposed to be called every 30 days

Proof Of Concept

Paste the following test into `Test_OCL_ZVE.sol`:

```
function testCanForwardYieldALot() public {
        address UNIV2_ROUTER = OCL_ZVE_UNIV2_DAI.router();
        deal(DAI, address(this), 10000);

        IERC20(DAI).safeApprove(UNIV2_ROUTER, 10000);
        IERC20(ZVE).safeApprove(UNIV2_ROUTER, 1001);

        IUniswapV2Router01(UNIV2_ROUTER).addLiquidity(address(DAI),
 ↪  address(ZVE), 10000, 1001, 0, 0, address(this), block.timestamp);

        uint256 count;
        while (block.timestamp > OCL_ZVE_UNIV2_DAI.nextYieldDistribution()) {
            OCL_ZVE_UNIV2_DAI.forwardYield();
            count++;
        }

        console.log(count);
    }
```

Code Snippet

https://github.com/sherlock-audit/2024-03-zivoe/blob/d4111645b19a1ad3ccc899bea073b6f19be04ccd/zivoe-core-foundry/src/lockers/OCL/OCL_ZVE.sol#L186 https://github.com/sherlock-audit/2024-03-zivoe/blob/d4111645b19a1ad3ccc899bea073b6f19be04ccd/zivoe-core-foundry/src/lockers/OCL/OCL_ZVE.sol#L287-L305

Tool used

Manual Review

Recommendation

Do not allow `OCL_ZVE::forwardYield()` to be called whenever the value of `nextYieldDistribution` is equal to 0. Also, another option is to set `nextYieldDistribution` to `block.timestamp + 30 days` instead of just incrementing it by `30 days`.

Discussion

coreggon11

If `pushToLockerMulti` was not called, then nothing will be forwarded.

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> medium, if `forwardYield` is called just after contract creation, then `nextYieldDistribution` becomes incorrect and `forwardYield` can be called at any time by anyone. The impact is limited, but the protocol functionality is broken (no time limit for calling `forwardYield`)

pseudonaut

Wouldn't `_forwardYield()` revert due to division by 0 if basis is 0 ?

```
function forwardYield() external {
        ...
        (uint256 amount, uint256 lp) = fetchBasis();
        if (amount > basis) { _forwardYield(amount, lp); }
```

Or rather `_forwardYield()` wouldn't even be called `if (amount > basis) { _forwardYield(amount, lp); }` ...

```
function _forwardYield(uint256 amount, uint256 lp) private nonReentrant {
        address ZVE = IZivoeGlobals_OCL_ZVE(GBL).ZVE();
        uint256 lpBurnable = (amount - basis) * lp / amount * (BIPS -
 ↪  compoundingRateBIPS) / BIPS;
```

panprog

@pseudonaut This is correct, `_forwardYield` will not be called, the point is that the last line of `forwardYield`:

```
nextYieldDistribution += 30 days;
```

will make `nextYieldDistribution = 30 days`, thus all future `forwardYield` will ignore timestamp check because `block.timestamp` will always be greater than `nextYieldDistribution`.

The issue is valid, keeping it medium.

spaceglideπrrr

Escalate

Issues which can be mitigated by correct admin configuration upon deployment are not valid according to Sherlock rules. Issue should be low.

sherlock-admin3

> Escalate
>
> Issues which can be mitigated by correct admin configuration upon deployment are not valid according to Sherlock rules. Issue should be low.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

samuraii77

> Escalate
>
> Issues which can be mitigated by correct admin configuration upon deployment are not valid according to Sherlock rules. Issue should be low.

Pushing into the locker upon deployment is in no way, shape or form, correct admin configuration. That is not even a configuration.

panprog

Agree with @samuraii77 , this can only be mitigated if admin pushes into locker in the same transaction as contract creation, which doesn't seem to be reasonable assumption as this should not be part of the deployment.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Zivoe/zivoe-core-foundry/pull/265

WangSecurity

As I understand from Zivoe's Deployment diagram on Figma, they indeed will initially call `push` on lockers (phase three) and only then `forwardYield` (phase four). Hence, I believe this report requires an admin acting as not intended (i.e. calling `forwardYield` before `pushToLockerMulti`), therefore, should be invalid.

Planning to reject an escalation and leave the issue as it is.

samuraii77

The issue does not rely on an admin acting not as intended. The `forwardYield()` function can be called by any user after the contract has been deployed which as visible by the diagram you provided happens at phase 1 and pushing happens all the way in phase 3. Basically, that means that a user has the time between phase 1 and phase 3 to just call `forwardYield()` in order for the vulnerability to happen. Issue should stay valid.

Also, I am not sure if I get you correctly but it seems like you are disputing my issue and agreeing with the escalation but then proceed to say that you will reject the escalation and keep the issue as it is, not sure if I interpreted your comment wrong.

WangSecurity

Yep, sorry for that two confusions here and I agree that indeed the locker is deployed earlier than push is called. Therefore, this issue indeed may occur.

Planning to reject the escalation and leave the issue as it is.

Evert0x

Result: Medium Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spaceglidererrr: rejected

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-5: When APR late rate is lower than APR, an OCC locker bullet loan borrower can pay way less interests by calling the loan

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/97

The protocol has acknowledged this issue.

Found by

0x73696d616f, Ironsidesec, KupiaSec, SilverChariot, saidam017, sl1, thank_you, y4y

Summary

A bullet loan borrower can pay less interests by calling callLoan at the end of payment period.

Vulnerability Detail

In `OCC_Modular` contract, the protocol can create loan offers, and users can accept them. The loan has two types, one being bullet, and the other being amortization. In the bullet loan, borrowers only need to pay back interests for each interval, and principle at the last term.

`amountOwed` returns the payment amount needed for each loan id:

```
function amountOwed(uint256 id) public view returns (
    uint256 principal, uint256 interest, uint256 lateFee, uint256 total
) {
    // 0 == Bullet.
    if (loans[id].paymentSchedule == 0) {
        if (loans[id].paymentsRemaining == 1) { principal =
↪   loans[id].principalOwed; }
    }
    // 1 == Amortization (only two options, use else here).
    else { principal = loans[id].principalOwed / loans[id].paymentsRemaining; }

    // Add late fee if past loans[id].paymentDueBy.
    if (block.timestamp > loans[id].paymentDueBy && loans[id].state ==
↪   LoanState.Active) {
        lateFee = loans[id].principalOwed * (block.timestamp -
↪   loans[id].paymentDueBy) *
            loans[id].APRLateFee / (86400 * 365 * BIPS);
    }
    interest = loans[id].principalOwed * loans[id].paymentInterval *
↪   loans[id].APR / (86400 * 365 * BIPS);
    total = principal + interest + lateFee;
}
```

And we see, there is a `lateFee` for any loans which is overdue. The later the borrower pays back the loan, the more late fees will be accumulated. Plus, the under writer role can always set the loan to default when it's way passed grace period. `callLoan` provides an option for borrowers to payback all he/she owes immediately and settles the loan. In this function, `amountOwed` is called once:

```
function callLoan(uint256 id) external nonReentrant {
    require(
        _msgSender() == loans[id].borrower ||
↪   IZivoeGlobals_OCC(GBL).isLocker(_msgSender()),
        "OCC_Modular::callLoan() _msgSender() != loans[id].borrower &&
↪   !isLocker(_msgSender())"
    );
    require(loans[id].state == LoanState.Active, "OCC_Modular::callLoan()
↪   loans[id].state != LoanState.Active");

    uint256 principalOwed = loans[id].principalOwed;
    (, uint256 interestOwed, uint256 lateFee,) = amountOwed(id);

    emit LoanCalled(id, principalOwed + interestOwed + lateFee, principalOwed,
↪   interestOwed, lateFee);

    // Transfer interest to YDL if in same format, otherwise keep here for 1INCH
↪   forwarding.
    if (stablecoin ==
↪   IZivoeYDL_OCC(IZivoeGlobals_OCC(GBL).YDL()).distributedAsset()) {
        IERC20(stablecoin).safeTransferFrom(_msgSender(),
↪   IZivoeGlobals_OCC(GBL).YDL(), interestOwed + lateFee);
    }
    else {
        IERC20(stablecoin).safeTransferFrom(_msgSender(), OCT_YDL, interestOwed
↪   + lateFee);
    }

    IERC20(stablecoin).safeTransferFrom(_msgSender(), owner(), principalOwed);

    loans[id].principalOwed = 0;
    loans[id].paymentDueBy = 0;
    loans[id].paymentsRemaining = 0;
    loans[id].state = LoanState.Repaid;
}
```

This means, only one interval's late fee is taken into account for this calculation. When the late fee rate is less than APR, and the payment is way overdue, it's possible for such borrower to skip a few interests and late fee payment.

```
function test_OCC_Late_Payment_Loan_ALL() public {
    uint256 borrowAmount = 20 * 10 ** 18;
    uint256 APR;
    uint256 APRLateFee;
    uint256 term;
    uint256 paymentInterval;
    uint256 gracePeriod;
    int8 paymentSchedule = 0;

    APR = 1000;
    APRLateFee = 500;
    term = 10;
    paymentInterval = options[1];
    gracePeriod = uint256(10 days) % 90 days + 7 days;
    mint("DAI", address(tim), 100 * 10 **18);
    hevm.startPrank(address(roy));
    OCC_Modular_DAI.createOffer(
        address(tim), borrowAmount, APR, APRLateFee, term, paymentInterval,
↪   gracePeriod, paymentSchedule
    );
    hevm.stopPrank();

    uint256 loanId = OCC_Modular_DAI.loanCounter() - 1;

    hevm.startPrank(address(tim));
    IERC20(DAI).approve(address(OCC_Modular_DAI), type(uint256).max);
    OCC_Modular_DAI.acceptOffer(loanId);
    (, , uint256[10] memory info) = OCC_Modular_DAI.loanInfo(loanId);
    hevm.warp(info[3] + (info[4] * term));
    uint256 balanceBefore = IERC20(DAI).balanceOf(address(tim));
    while (info[4] > 0) {
        OCC_Modular_DAI.makePayment(loanId);
        (, , info) = OCC_Modular_DAI.loanInfo(loanId);
    }
    uint256 balanceAfter = IERC20(DAI).balanceOf(address(tim));
    uint256 diff = balanceBefore - balanceAfter;
    console.log("paid total when payments are late for each interval:", diff);
    console.log("total interests:", diff - borrowAmount);
    hevm.stopPrank();
}
```

```
function test_OCC_Normal_Payment_Loan() public {
    uint256 borrowAmount = 20 * 10 ** 18;
    uint256 APR;
    uint256 APRLateFee;
    uint256 term;
    uint256 paymentInterval;
    uint256 gracePeriod;
    int8 paymentSchedule = 0;

    APR = 1000;
    APRLateFee = 500;
    term = 10;
    paymentInterval = options[1];
    gracePeriod = uint256(10 days) % 90 days + 7 days;
    mint("DAI", address(tim), 100 * 10 ** 18);
    hevm.startPrank(address(roy));
    OCC_Modular_DAI.createOffer(
        address(tim), borrowAmount, APR, APRLateFee, term, paymentInterval,
 ↪  gracePeriod, paymentSchedule
    );
    hevm.stopPrank();

    uint256 loanId = OCC_Modular_DAI.loanCounter() - 1;

    hevm.startPrank(address(tim));
    IERC20(DAI).approve(address(OCC_Modular_DAI), type(uint256).max);
    OCC_Modular_DAI.acceptOffer(loanId);
    (, , uint256[10] memory info) = OCC_Modular_DAI.loanInfo(loanId);
    hevm.warp(info[3]);
    uint256 balanceBefore = IERC20(DAI).balanceOf(address(tim));
    while (info[4] > 0) {
        OCC_Modular_DAI.makePayment(loanId);
        (, , info) = OCC_Modular_DAI.loanInfo(loanId);
        hevm.warp(info[3]);
    }
    uint256 balanceAfter = IERC20(DAI).balanceOf(address(tim));
    uint256 diff = balanceBefore - balanceAfter;
    console.log("paid total when loan is solved normally:", diff);
    console.log("total interests:", diff - borrowAmount);
    hevm.stopPrank();
}

function test_OCC_Late_Call_Loan() public {
    uint256 borrowAmount = 20 * 10 ** 18;
    uint256 APR;
    uint256 APRLateFee;
    uint256 term;
    uint256 paymentInterval;
    uint256 gracePeriod;
    int8 paymentSchedule = 0;

    APR = 1000;
    APRLateFee = 500;
    term = 10;
    paymentInterval = options[1];
    gracePeriod = uint256(10 days) % 90 days + 7 days;
    mint("DAI", address(tim), 100 * 10 ** 18);
    hevm.startPrank(address(roy));
    OCC_Modular_DAI.createOffer(
        address(tim), borrowAmount, APR, APRLateFee, term, paymentInterval,
↪   gracePeriod, paymentSchedule
    );
    hevm.stopPrank();

    uint256 loanId = OCC_Modular_DAI.loanCounter() - 1;

    hevm.startPrank(address(tim));
    IERC20(DAI).approve(address(OCC_Modular_DAI), type(uint256).max);
    OCC_Modular_DAI.acceptOffer(loanId);
    (, , uint256[10] memory info) = OCC_Modular_DAI.loanInfo(loanId);
    hevm.warp(info[3] + (info[4] * term));
    uint256 balanceBefore = IERC20(DAI).balanceOf(address(tim));
    OCC_Modular_DAI.callLoan(loanId);
    uint256 balanceAfter = IERC20(DAI).balanceOf(address(tim));
    uint256 diff = balanceBefore - balanceAfter;
    console.log("paid total when use `callLoan` at the end:", diff);
    console.log("total interests:", diff - borrowAmount);
    hevm.stopPrank();
}
```

In the above test cases, all three of them will have the same borrower, and borrow the same loan, with same details and everything. One of them simulating when a borrower pays all charges normally till the end of term, another one waits till the very end to pay back the loan with late fees, and the last one also wait till the end, except calls `callLoan` to settle the loan instead of normally paying back each interval's amount.

After running the test cases, the following will be logged:

As we can see, while `callLoan` also needs to pay the late fee penalty, it still charges way less than normally paying back the loan. This makes a borrower being able to skip a few interests fee, with the cost of little late fees.

Impact

The PoC provided above is certainly an exaggerated edge case, but it's also possible when late fees are aribitrary, as long as the loan is not set to default by under writers, the borrower can skip paying quite some interest fees by exploiting this at the cost of a few late fees. This is more likely to happen when intervals are set to 7 days, as the minimum grace period is 7 days.

Code Snippet

```
function callLoan(uint256 id) external nonReentrant {
    require(
        _msgSender() == loans[id].borrower ||
↪   IZivoeGlobals_OCC(GBL).isLocker(_msgSender()),
        "OCC_Modular::callLoan() _msgSender() != loans[id].borrower &&
↪   !isLocker(_msgSender())"
    );
    require(loans[id].state == LoanState.Active, "OCC_Modular::callLoan()
↪   loans[id].state != LoanState.Active");

    uint256 principalOwed = loans[id].principalOwed;
    (, uint256 interestOwed, uint256 lateFee,) = amountOwed(id);

    emit LoanCalled(id, principalOwed + interestOwed + lateFee, principalOwed,
↪   interestOwed, lateFee);

    // Transfer interest to YDL if in same format, otherwise keep here for 1INCH
↪   forwarding.
    if (stablecoin ==
↪   IZivoeYDL_OCC(IZivoeGlobals_OCC(GBL).YDL()).distributedAsset()) {
        IERC20(stablecoin).safeTransferFrom(_msgSender(),
↪   IZivoeGlobals_OCC(GBL).YDL(), interestOwed + lateFee);
    }
    else {
        IERC20(stablecoin).safeTransferFrom(_msgSender(), OCT_YDL, interestOwed
↪   + lateFee);
    }

    IERC20(stablecoin).safeTransferFrom(_msgSender(), owner(), principalOwed);

    loans[id].principalOwed = 0;
    loans[id].paymentDueBy = 0;
    loans[id].paymentsRemaining = 0;
    loans[id].state = LoanState.Repaid;
}
```

Tool used

Manual Review, foundry

Recommendation

Prohibits bullet loan borrowers from calling `callLoan` when the loan is late and still has more than 1 intervals to finish.

Discussion

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> medium, if there are more than 1 interest payments missed by borrower, then `callLoan` takes only 1 period payment, allowing borrower to skip paying the other periods interest and lateFee payments.

pseudonaut

This is not a valid issue - calling a loan is much different than making a payment as it requires the full amount of principal vs. (in the case of bullet loans) a single payment with interest only

panprog

Yes, it's much different from making a payment, but it still allows to bypass paying additional interests/late fees, for example, when only a few payment periods are remaining and it makes sense to simply `callLoan` instead of doing last 2-3 interest and latefee payments at the expense of protocol / depositors.

Keeping this as medium.

panprog

Sponsor response:

> intended functionality, user has option to callLoan at any point in time, and if you're saying they have late-fee's than theoretically the loan could be defaulted and likely would at that point preventing callLoan

panprog

This is from docs:

> In some cases, the grace period may be longer than the payment interval, and the borrower may miss several loan payments before a loan

> enters default. In such an event, the borrower must resolve each missed payment before late fees stop accruing.

So it's possible that grace period is longer than payment interval. Example: payment interval = 7 days, grace period = 28 days. Borrower can simply stop paying 28 days before the loan is due. At the end of the loan he will have 4 missed payments, at which point he simply `callLoan` and pay only 1 missed payment + late fees from it, skipping paying the other 3 missed payments and late fees from them. Depending on late fees this can be cheaper for borrower than paying all 4 payments on time. Scenario A (paying on time): 28 days * APR payments Scenario B (calling loan in the end): 7 days * APR + 28 days * lateFee APR

If lateFee is less than APR, then borrower is better off skipping the last 4 payments and doing `callLoan` in the end.

Keeping this medium.

Issue M-6: Rewards are calculated as distributed even if there are no stakers, locking the rewards forever

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/113

The protocol has acknowledged this issue.

Found by

0x73696d616f, BoRonGod, Ironsidesec, SUPERMAN_I4G, Tendency, dimulski, marchev, rbserver

Summary

The `ZivoeRewards.sol` and the `ZivoeRewardsVesting.sol` contracts are a fork of the Synthetix rewards distribution contract, with slight modifications. The contract logic keeps track of the total time passed, as well as the cumulative rate of rewards that have been generated for each token used for rewards, so that for every user, it just has to keep track of the last timestamp and last rate at the time of the last reward withdrawal, stake, or unstake in order to calculate the rewards owed since the user began staking. The code special-cases the scenario where there are no users, by not updating the cumulative rate when the _totalSupply is zero, but it does not include such a condition for the tracking of the timestamp. Because of this, even when there are no users staking, the accounting logic still thinks funds were being dispersed during that timeframe (because the starting timestamp is updated), which means the funds effectively are distributed to nobody, rather than being saved for when there is someone to receive them. And will be locked in the contract forever. One of the modifications is this line in the depositReward() function.

```
IERC20(_rewardsToken).safeTransferFrom(_msgSender(), address(this), reward);
```

Contrary to the Synthetix implementation, Zivo requires each time reward is deposited to the contact, the reward amount to be transferred in the same transaction. However if a reward is deposited but there are no stakers, the reward that should have been distributed for the period until the first user stakes, will be locked in the contract forever, and won't be able to be added to the rewards for the next reward period because of the above code snippet.

Vulnerability Detail

Gist After following the steps in the above linked gist add the following test to the `AuditorTests.t.sol` contract:

```
function test_LockedRewards() public {
    vm.startPrank(ZVL);
    zivoeToken.transfer(alice, 5_000_000e18);
    uint256 duration = 30 days;
    stZVE.addReward(address(mockUSDC), duration);
    vm.stopPrank();

    vm.startPrank(simulateYLD);
    mockUSDC.mint(simulateYLD, 50_000e6); // this represent 50_000 USDC tokens
↪   to be distributed in the next 30 days
    mockUSDC.approve(address(stZVE), type(uint256).max);
    stZVE.depositReward(address(mockUSDC), 50_000e6);
    skip(172_800); /// @notice two days pass, before anybody stakes in the
↪   contract
    vm.stopPrank();

    vm.startPrank(alice);
    console2.log("USDC balance of alice before staking: ",
↪   mockUSDC.balanceOf(alice));
    console2.log("USDC balance of stZVE contract: ",
↪   mockUSDC.balanceOf(address(stZVE)));
    zivoeToken.approve(address(stZVE), type(uint256).max);
    stZVE.stake(5_000_000e18);
    skip(duration);
    stZVE.fullWithdraw();
    console2.log("USDC balance of alice after staking for 30 days: ",
↪   mockUSDC.balanceOf(alice));
    console2.log("USDC balance of stZVE contract, when users have withdrawn
↪   everything for the first period: ", mockUSDC.balanceOf(address(stZVE)));
    console2.log("USDC balance of stZVE contract, when users have withdrawn
↪   everything for the first period normalized: ",
↪   mockUSDC.balanceOf(address(stZVE)) / 1e6);
    console2.log("zivoToken balance of alice after unstaking: ",
↪   zivoeToken.balanceOf(alice));
    vm.stopPrank();
}
```

```
Logs:
  USDC balance of alice before staking:  0
  USDC balance of stZVE contract:  50000000000
  USDC balance of alice after staking for 30 days:  46665000000
  USDC balance of stZVE contract, when users have withdrawn everything for the
↪   first period:  3335000000
  USDC balance of stZVE contract, when users have withdrawn everything for the
↪   first period normalized:  3335
```

```
  zivoToken balance of alice after unstaking:  5000000000000000000000000
```

As can be seen from the above logs **3335 USDC tokens** will be locked forever.

To run the test use: `forge test -vvv --mt test_LockedRewards`

Impact

If the depositReward() function is called prior to there being any users staking, the funds that should have gone to the first stakers will instead accrue to nobody, and be locked in the contract forever.

Code Snippet

https://github.com/sherlock-audit/2024-03-zivoe/blob/main/zivoe-core-foundry/src/ZivoeRewards.sol#L228-L243

Tool used

Manual Review & Foundry

Recommendation

Remove the `safeTransfer()` from the depositReward() function, and instead check if there are enough reward tokens already in the contract. Something similar to the Synthetix implementation.

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> borderline medium/low. If there are no stakers and depositReward is called, the deposit amount might be lost since it isn't distributed to anybody.

panprog

Keeping this medium as the loss of funds can happen, although the probability of this is very low, but still possible.

spaceglider rrr

Escalate

It would be an admin/ user mistake to deposit rewards when there are no stakers. Issue should be low.

sherlock-admin3

> Escalate
>
> It would be an admin/ user mistake to deposit rewards when there are no stakers. Issue should be low.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

panprog

The rewards are sent from the lockers into ZivoeYDL, which then calls `depositReward` in the `distributeYield` function, which can be called by any user, thus the rewards might be lost without any admin interaction, only from (any) user interaction. And it's not user depositing rewards, it's user forcing accumulated reward to be deposited.

WangSecurity

Need clarification how it can distribute rewards if there are no stakers. As I know the users deposit into tranches and their funds are used for either loans or investing in other protocols, correct? For depositing into tranches, the users receive ZVE token and then can stake it.

But in the edge case, no one staked their ZVE token, but the funds from tranches were invested in some protocol and accrued yield. That yield will be distributed among users, but since no one staked their ZVE, the funds are stuck inside the contract and cannot be withdraw by anyone?

panprog

@WangSecurity The following are `ZivoeRewards` contracts (from `ZivoeGlobals`):

```
address public stJTT;       /// @dev The ZivoeRewards ($stJTT) contract.
address public stSTT;       /// @dev The ZivoeRewards ($stSTT) contract.
address public stZVE;       /// @dev The ZivoeRewards ($stZVE) contract.
```

When user deposits into tranche, he's given tranche tokens (STT or JTT). But in order to get the rewards, user will have to stake his token into stSTT/stJTT, which are ZivoeRewards. When the `ZivoeYDL` distributes yield via `distributeYield`, it deposits this yield as reward into stSTT/stJTT based on emaSTT/emaJTT, which are averages of the adjusted balances (not staked tokens). So if for whatever reason all users unstake their stSTT or stJTT tokens, it is possible that there is some distribution accumulated, which is deposited into stSTT or stJTT, but there are no stakers.

This is not a concern with stZVE, because it splits the yield between stZVE / vestZVE based on staked tokens total supply, so if stZVE total supply == 0, it will receive 0 reward.

WangSecurity

In that case, the likelihood of that scenario is extremely low, but no guarantee that it doesn't happen, leading to a loss of funds if the issue occurs.

Planning to reject the escalations and leave the issue as it is.

Evert0x

Result: Medium Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spacegliderrrr: rejected

Issue M-7: EMA Data Point From Unlock Is Discarded

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/201

Found by

JigglypuffAndPikachu, Tendency, lemonmon, pseudoArtist

Summary

The `ema` calculation fails to incorporate the `emaJTT` and `emaSTT` amounts which are initally set in `unlock`.

Vulnerability Details

Let's carefully consider the EMA calculation. Firstly during the unlock, the `emaSTT` and `emaJTT` are set to the current values. This acts as our first data point. Since this is the only data point, this doesn't have any averaging which is why `MATH.ema()` is not called.

Now, after unlocking consider when `distributeYield` is called for the first time after unlocking. `emaSTT` and `emaJTT` should incorporate the first data point (which was recorded during unlock) with the new `totalSupply` of `STT` and `JTT`.

Now let's show why the contract will actually ignore the `emaSTT`/JTT set during unlock during the first yield distribution:

The `distributionCounter` is `0` when `distributeYield` is called, but due to the line `distributionCounter += 1;`, `1` is passed as the 3rd parameter to the `ema` formula in `emaSTT = MATH.ema(emaSTT, aSTT, retrospectiveDistributions.min(distributionCounter));`

When `N == 1` in the `ema` formula, the function will return only `eV`, which is only the latest data point. It completely ignores `bV` which is the data point during the unlock:

```
function ema(uint256 bV, uint256 cV, uint256 N) external pure returns (uint256
↪   eV) {

    assert(N != 0);

    uint256 M = (WAD * 2).floorDiv(N + 1); //When N = 1, M = WAD

    eV = ((M * cV) + (WAD - M) * bV).floorDiv(WAD); //Substituting M = WAD, eV =
↪   cV

}
```

Impact

Incorrect EMA calculation which leads to incorrect yield distribution

Code Snippet

https://github.com/sherlock-audit/2024-03-zivoe/blob/main/zivoe-core-foundry/src/ZivoeYDL.sol#L213-L310

Tool used

Manual Review

Recommendation

The `distributionCounter` should actually be `2` rather than 1 during the first pass into the `MATH.ema()` formula. However then the varibale would not really correspond to the number of times `distribution` was called, so it might need to be renamed.

Discussion

pseudonaut

Valid, distributionCounter must be set to 1 initially

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> borderline low/medium, while true, the impact is questionable as there can be situations when current method is better than fixed one, so this is basically a choice of N=1 or N=2 - both are valid, just whatever developer prefers to better fit the protocol needs.

panprog

Keeping this medium as sponsor confirmed it's valid (thus considering that N=2 is the intended value, but it's N=1 right now). The impact then is unintended values used in further calculations.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Zivoe/zivoe-core-foundry/pull/252

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-8: `ZivoeTranches#rewardZVEJuniorDeposit` function miscalculates the reward when the ratio traverses lower/upper bound.

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/232

The protocol has acknowledged this issue.

Found by

AMOW, amar, dany.armstrong90

Summary

`ZivoeTranches#rewardZVEJuniorDeposit` function miscalculates the reward when the ratio traverses lower/upper bound. The same issue also exists in the `ZivoeTranches#rewardZVESeniorDeposit` function.

Vulnerability Detail

`ZivoeTranches#rewardZVEJuniorDeposit` function is the following.

```
    function rewardZVEJuniorDeposit(uint256 deposit) public view returns
↪   (uint256 reward) {

        (uint256 seniorSupp, uint256 juniorSupp) =
↪   IZivoeGlobals_ZivoeTranches(GBL).adjustedSupplies();

        uint256 avgRate;    // The avg ZVE per stablecoin deposit reward, used
↪   for reward calculation.

        uint256 diffRate = maxZVEPerJTTMint - minZVEPerJTTMint;

        uint256 startRatio = juniorSupp * BIPS / seniorSupp;
        uint256 finalRatio = (juniorSupp + deposit) * BIPS / seniorSupp;
213:    uint256 avgRatio = (startRatio + finalRatio) / 2;

        if (avgRatio <= lowerRatioIncentiveBIPS) {
216:        avgRate = maxZVEPerJTTMint;
        } else if (avgRatio >= upperRatioIncentiveBIPS) {
218:        avgRate = minZVEPerJTTMint;
        } else {
220:        avgRate = maxZVEPerJTTMint - diffRate * (avgRatio -
↪   lowerRatioIncentiveBIPS) / (upperRatioIncentiveBIPS -
↪   lowerRatioIncentiveBIPS);
```

```
        }

223:    reward = avgRate * deposit / 1 ether;

        // Reduce if ZVE balance < reward.
        if
↪   (IERC20(IZivoeGlobals_ZivoeTranches(GBL).ZVE()).balanceOf(address(this)) <
↪   reward) {
            reward =
↪   IERC20(IZivoeGlobals_ZivoeTranches(GBL).ZVE()).balanceOf(address(this));
        }
    }
```

Here, let us assume that `lowerRatioIncentiveBIPS = 1000`, `upperRatioIncentiveBIPS = 2500`, `minZVEPerJTTMint = 0`, `maxZVEPerJTTMint = 0.4 * 10 ** 18`, `seniorSupp = 10000`.

Let us consider the case of `juniorSupp = 0` where the ratio traverses the lower bound.

Example 1: Assume that the depositor deposit `2000` at a time. Then `avgRatio = 1000` holds in `L213`, thus `avgRate = maxZVEPerJTTMint = 0.4 * 10 ** 18` holds in `L216`. Therefore `reward = 0.4 * deposit = 800` holds in `L223`.

Example 2: Assume that the depositor deposit `1000` twice. Then, since `avgRate = 500 < lowerRatioIncentiveBIPS` holds for the first deposit, `avgRate = 0.4 * 10 ** 18` holds in `L216`, thus `reward = 400` holds. Since `avgRate = 1500 > lowerRatioIncentiveBIPS` holds for the second deposit, `avgRate = 0.3 * 10 ** 18` holds in `L220`, thus `reward = 300` holds. Finally, the total sum of rewards for two deposits are `400 + 300 = 700`.

This shows that the reward of the case where all assets are deposited at a time is larger than the reward of the case where assets are divided and deposited twice. In this case, the protocol gets loss of funds.

Likewise, in the case where the ratio traverses the upper bound, the reward of one time deposit will be smaller than the reward of two times deposit and thus the depositor gets loss.

The same issue also exists in the `ZivoeTranches#rewardZVESeniorDeposit` function.

Impact

When the ratio traverses the lower/upper bound in `ZivoeTranches#rewardZVEJuniorDeposit` and `ZivoeTranches#rewardZVESeniorDeposit` functions, the amount of reward will be larger/smaller than it should be. Thus the depositor or the protocol will get loss of funds.

Code Snippet

https://github.com/sherlock-audit/2024-03-zivoe/blob/main/zivoe-core-foundry/src/ZivoeTranches.sol#L215-L221

Tool used

Manual Review

Recommendation

Modify the functions to calculate the reward dividing into two portion when the ratio traverses the lower/upper bound, which is similar to the case of Example 2.

Discussion

pseudonaut

Yes if the tranche ratio's swing very far out it will allow for calculations of this nature - known, not of concern

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> borderline low/medium, while true, the impact is not very large and can be considered protocol's intended choice to simplify the curve. From protocol's standpoint this is simply incentive to deposit and slightly more or less in specific circumstances is not that important.

panprog

Downgrading to Low.

This is known to developers and I consider this a design decision to simplify the calculations / save gas. The user can be informed about such behaviour and can do multiple or single transaction depending on what's more beneficial. No funds loss from protocol's standpoint (simply slightly different ratio of rewards distribution which doesn't influence anything)

armormadeofwoe

Continuing discussion here.
Here is a graph of the juniorReward linear decrease with respect to the min/max reward thresholds. As mentioned in this issue and its' 2 dups - a 2-deposit split where the initial one is crafted to not cross the max threshold will always yield more

rewards than going at once. I believe this issue's math paired with a visualization should be sufficient. Let me know your thoughts.

Issue M-9: Time calculation issues with exponential decay

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/282

The protocol has acknowledged this issue.

Found by

AMOW, SilverChariot

Summary

OCE_ZVE is a locker that distributes rewards based on a exponential decay model. When a distribution starts, the whole balance of the contract should be unclaimable. As time moves forward, more and more of it should be unlocked for distributing. The calculations are made using the `lastDistribution` variable which is initially set in the constructor and then changed on each distribution. This is problematic because the "idle" periods where no distribution is happening will be considered as passed time when a real distribution starts.

Vulnerability Detail

In the constructor, the lastDistribution variable is set to `block.timestamp`.

```
lastDistribution = block.timestamp;
```

When forwardEmissions() gets called, the calculation `block.timestamp - lastDistribution` will return a large value because the timer has started at the time of deployment.

```
function forwardEmissions() external nonReentrant {
    uint zveBalance =
↪   IERC20(IZivoeGlobals_OCE_ZVE(GBL).ZVE()).balanceOf(address(this));
    _forwardEmissions(zveBalance - decay(zveBalance, block.timestamp -
↪   lastDistribution));
    lastDistribution = block.timestamp;
}
```

As we can see in the Figma file, the OCE locker will be deployed at `Phase One` and funding will start after ITO ends, which is at least 30 days.

This results in a wrong calculation and instead of decaying, a big amount of the rewards can be forwarded as soon as the distribution starts.

The issue persists for further distributions also. If distribution 1 ends on 1 January and the Zivoe team decides to start distribution 2 on 1 July, the rewards for 6 months will be claimable from the very beginning. Clearing the timestamp before a distribution starts is not an option because it requires at least `100e18` assets to be forwarded.

```
require(amount >= 100 ether, "OCE_ZVE::_forwardEmissions amount < 100 ether");
```

Impact

Instead of decaying, a big part of the rewards is claimable from the beginning.

Code Snippet

PoC for Test_OCE_ZVE

```
function test_OCE_timer() public {
    hevm.warp(block.timestamp + 365 days);
    deal(address(ZVE), address(OCE_ZVE_Live), 20000e18);
    OCE_ZVE_Live.forwardEmissions();
}
```

Tool used

Foundry

Recommendation

Add a guarded function that start the distribution by updating the timestamp.

```
function startDistribution() external {
         require(
            _msgSender() == IZivoeGlobals_OCE_ZVE(GBL).TLC(),
            "OCE_ZVE::startDistribution() _msgSender() !=
↪   IZivoeGlobals_OCE_ZVE(GBL).TLC()"
        );
        lastDistribution = block.timestamp;
}
```

Discussion

pseudonaut

forwardEmissions can be called prior to reset, not an issue

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> medium, OCE_ZVE forwardEmission will start emission as if some large time has already passed due to absence of "start emission" function (to reset lastDistribution timestamp). While it's possible to reset it by calling `forwardEmission`, it requires a minimum of 100e18 tokens, which will have to be distributed to reset the timestamp, which is still a loss of funds. As this is mostly a 1-time action, the impact is limited.

panprog

Keeping this medium, because a distribution of 100e18 tokens is still required to reset the `lastDistribution` timestamp.

Issue M-10: OCL_ZVE.sol::forwardYield relies on manipulable Uniswap V2 pool reserves leading to theft of funds

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/296

The protocol has acknowledged this issue.

Found by

0xpiken, 0xvj, AllTooWell, Audinarey, DPS, Drynooo, JigglypuffAndPikachu, Maniacs, SilverChariot, cergyk, lemonmon, saidam017, t0x1c

Summary

`OCL_ZVE.sol::forwardYield` is used to forward yield in excess of the basis but it relies on Uniswap V2 pools that can be manipulable by an attacker to set `basis` to a very small amount and steal funds.

Vulnerability Detail

`fetchBasis` is a function which returns the amount of pairAsset (USDC) which is claimable by burning the balance of LP tokens of OCL_ZVE

`fetchBasis` is called first in `forwardYield` in order to determine the yield to distribute:

OCL_ZVE.sol#L301-L303

```
>>    (uint256 amount, uint256 lp) = fetchBasis();
        if (amount > basis) { _forwardYield(amount, lp); }
        (basis,) = fetchBasis();
```

If the returned `amount` is `<= basis` (negative yield) no yield is forwarded and basis is updated to the low `amount` value

By manipulating the Uniswap V2 pool with a flashloan, an attacker can get the `uint256 amount` value to be returned by `OCL_ZVE.sol::fetchBasis` to be very small and set `basis` to this very small value:

OCL_ZVE.sol#L301-L303

```
        if (amount > basis) { _forwardYield(amount, lp); }
>>      (basis,) = fetchBasis();
```

The next call (30 days later) to `OCL_ZVE.sol::forwardYield` will forward much more yield (in excess of `basis`) than it should, leading to a loss of funds for the protocol.

Scenario

1. Attacker buys a very large amount of USDC in the Uniswap V2 pool `ZVE/pairAsset` (can use a flash-loan if needed)
2. Attacker calls `OCL_ZVE.sol::forwardYield`

- `OCL_ZVE.sol::fetchBasis` returns an incorrect and very small value for `amount`:
- No yield is forwarded since `amount < basis`
- `OCL_ZVE.sol::forwardYield` sets `basis` to `amount`

3. Attacker backruns the calls to `OCL_ZVE.sol::forwardYield` with the sell of his large buy in the Uniswap V2 pool
4. 30 days later, Attacker calls `OCL_ZVE.sol::forwardYield` and steal funds in excess of `basis`

Impact

Loss of funds for the protocol.

Code Snippet

- https://github.com/sherlock-audit/2024-03-zivoe/blob/d4111645b19a1ad3ccc899bea073b6f19be04ccd/zivoe-core-foundry/src/lockers/OCL/OCL_ZVE.sol#L302-L303
- https://github.com/sherlock-audit/2024-03-zivoe/blob/d4111645b19a1ad3ccc899bea073b6f19be04ccd/zivoe-core-foundry/src/lockers/OCL/OCL_ZVE.sol#L311-L330

Tool used

Manual Review

Recommendation

Consider reverting during a call to `forwardYield` if `amount <= basis`

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> medium, allows to distribute almost entire principal as if it's yield, leaving the locker with almost no funds. Since `forwardYield` will usually be called by keepers, who will use flashbots thus making frontrunning out of scope, this is medium. User can still call `forwardYield` but only if keeper didn't call it in the 12 hours prior to end of period, so this requires additional conditions.

panprog

Keeping this medium, because `forwardYield` can still be called by user in some circumstances, thus making the attack possible. Even though the distribution is within the protocol, it still distributes inflated amounts to tranches and/or residual recepients, causing accounting issues for the protocol.

panprog

Sponsor response:

> it's protected by keeper calls for 12 hours prior and if not allows public accessibility, obviously mev-mitiation is implemented with front-running in mind and then allows public accessibility, intended functionality

panprog

It is still possible that user calls this with front-running, even though the probability is very low (keeper has to be down for 12 hours, there might be high network congestion for 12+ hours with keeper being unable to submit transaction due to high gas price etc). This is Medium by Sherlock rules (possible, but very low probability of happening).

spacegliderrrr

Escalate

According to Sherlock rules, admins (keepers in this case) are expected to behave properly and not miss their 12hrs window to call this function. Furthermore, this issue would require keepers to miss the 12hrs window 2 months in a row (as if it just happens only once, issue can be mitigated the following month). Issue should be low.

sherlock-admin3

> Escalate
>
> According to Sherlock rules, admins (keepers in this case) are expected to behave properly and not miss their 12hrs window to call this function. Furthermore, this issue would require keepers to miss the 12hrs window 2 months in a row (as if it just happens only once, issue can be mitigated the following month). Issue should be low.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

ZdravkoHr

Why is it required that keepers don't call it 2 times?

spaceglidderrrr

First time, it would simply not distribute yield (and set basis to very low value). Next month if keepers call it, they can adjust it so it distributes just the right amount.

If only 1 attack call is made, biggest impact is rolling over the yield for next month (but it is not lost in any way).

ZdravkoHr

Can't the amount be manipulated in the opposite direction instead to cause distribution on the first call?

panprog

The keeper might be down for factors outside of admin control, so I can't consider "calling within 12 hours" a proper keeper behaviour. The remark about 2 calls is incorrect: only 1 is enough. For example, current basis = 10. User manipulates basis to be 1. During the next distribution user doesn't have to manipulate it, normal distribution will happen with current basis = 10 (or whatever is current basis) and distribute most of the funds as if it's yield.

panprog

> Can't the amount be manipulated in the opposite direction instead to cause distribution on the first call?

It's possible, but attacker will lose a lot of funds for such manipulation, so it's very costly to do so, reduction of basis is almost free for attacker, causing the same/bigger impact just later.

WangSecurity

Firstly, I agree that keeper is indeed may be down and miss the call to `forwardYield`. The probability is low, but as I understand, only one missed call by the keeper is enough for this scenario to occur, leading to a loss of funds for the protocol. Hence, I believe medium is appropriate.

Planning to reject the escalation and leave the issue as it is.

Evert0x

Result: Medium Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spaceglidерrrr: rejected

Issue M-11: ZivoeYDL::distributeYield yield distribution is flash-loan manipulatable

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/298

The protocol has acknowledged this issue.

Found by

cergyk

Summary

`ZivoeYDL::distributeYield` is used to "distributes available yield within this contract to appropriate entities" but it relies on the tokens `totalSupply()` which can be manipulable through a flashloan.

Vulnerability Detail

`ZivoeYDL::distributeYield` relies on `stZVE().totalSupply()` to distribute protocol earnings and residual earnings:

ZivoeYDL.sol#L241-L310

```
        // Distribute protocol earnings.
...
            else if (_recipient == IZivoeGlobals_YDL(GBL).stZVE()) {
                uint256 splitBIPS = (
>>                  IERC20(IZivoeGlobals_YDL(GBL).stZVE()).totalSupply() * BIPS
                ) / (
                    IERC20(IZivoeGlobals_YDL(GBL).stZVE()).totalSupply() +
                    IERC20(IZivoeGlobals_YDL(GBL).vestZVE()).totalSupply()
                );
...
        // Distribute residual earnings.
...
                else if (_recipient == IZivoeGlobals_YDL(GBL).stZVE()) {
                    uint256 splitBIPS = (
>>                      IERC20(IZivoeGlobals_YDL(GBL).stZVE()).totalSupply() *
 ↪  BIPS
                    ) / (
                        IERC20(IZivoeGlobals_YDL(GBL).stZVE()).totalSupply() +
                        IERC20(IZivoeGlobals_YDL(GBL).vestZVE()).totalSupply()
                    );
...
```

This can be abused by an attacker by buying then staking a very large amount of `ZVE` right before calling `ZivoeYDL::distributeYield` (a flashloan can be used) in order to game the system and collect a lot more distributed yields than he should be entitled to.

Scenario

1. Attacker buys and stakes a very large amount of `ZVE` through a flashloan
2. Attacker calls `ZivoeYDL::distributeYield`
3. Attacker collects a very large amount of distributed yields
4. Attacker withdraws and sells back his `ZVE` tokens effectively stealing undeserved yields

Impact

A user can systematically claim a big chunk of the rewards reserved to `stZVE` and `vestZVE` in `ZivoeYDL` by using a flash loan

Code Snippet

- https://github.com/sherlock-audit/2024-03-zivoe/blob/d4111645b19a1ad3ccc899bea073b6f19be04ccd/zivoe-core-foundry/src/ZivoeYDL.sol#L241-L310

Tool used

Manual Review

Recommendation

Use a minimal time locking for stZVE such as it would not be possible to stake and unstake all in one block, or use past (1 block in the past) total supply to claim rewards

Discussion

pseudonaut

Not valid

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> medium. The scenario presented isn't actually possible, because staking reward is deposited to be distributed over the long period of time, so the attacker won't earn anything from staking for 0 blocks. However, it still allows to manipulate the ratio of stZVE/vestZVE reward allocation, so it's possible to inflate reward to stZVE, then withdraw, but stZVE will receive inflated reward (where attacker can have a significant amount of token staked to benefit from increased yield). This is at the expense of vestZVE which will receive deflated reward.

panprog

Keeping this medium.

While the scenario is not possible as described, it still describes a valid issue, attack vector and core reason (manipulation to inflate stZVE's share of distribution).

panprog

Sponsor response:

> there's no economically feasible way to make this occur, given restricted circulating supply (<10% circulating for a number of years, with 30% in DAO and 60% in vestZVE, slippage/fees on DEXs and likely supply lowered due to CEX trading) and it implies that user maintains amount being staked after the flash-loan (separate from whatever was used to manipulate it) and doesn't even account for other users coming in over next 30 days and diluting any additional yield earned

panprog

You don't need a lot of ZVE to impact the reward distribution. For example, the ratio is 1% in stZVE and 99% in vestZVE. If you buy 1% of ZVE, stake it, call distributeYield, then unstake and sell back ZVE, the ratio is now 2% in stZVE, 98% in vestZVE, so a double rate for stZVE for the next 30 days. The loss for vestZVE is not large, but still loss nonetheless.

Keeping this medium.

0x73696d616f

Escalate

The issue described should be invalid as it is the intended behaviour of the protocol. Since the user has to stake to get the rewards for some time, he would have done the regular operation for users. The user is not getting more rewards for himself than any user that legitimately staked before. He is just staking closer to the deadline. Keep in mind that the protocol is always handing out the same APR (unless not enough liquidity is available) to all stakers, so in practice other users would not even lose from this.

sherlock-admin3

> Escalate
>
> The issue described should be invalid as it is the intended behaviour of the protocol. Since the user has to stake to get the rewards for some time, he would have done the regular operation for users. The user is not getting more rewards for himself than any user that legitimately staked before. He is just staking closer to the deadline. Keep in mind that the protocol is always handing out the same APR (unless not enough liquidity is available) to all stakers, so in practice other users would not even lose from this.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

panprog

The issue is that a 1-transaction inflated staked amount allows to inflate stakers distribution at the loss of vesters distribution.

WangSecurity

I'm not sure I understand how it effects the yield after we call `distributeYield`. If we take example from this comment:

The ratio is 1% stZVE and 99% vstZVE. The attacker stakes ZVE, so it's now 2% for stZVE, call disitrbute yield, then unstake ZVE, so it's again 1%.

The timelock to call distributeYield passes and we call it again. We firstly calculate the ratio, which will be 1% stZVE and 99% vstZVE (I think I miss something here) and then the yield is distributed. So the main question is are these inflated ratios (2% for stZVE) actually saved upon the next time we call `distributeYield`?

Or the problem is that we can front-run the call to `ditributeYield` every time to get more rewards?

(Sorry if silly questions, but the phrase "double rate for stZVE for the next 30 days" really confuses me, since the ratio is calculated right before transfering the rewards)

0x73696d616f

Hey @WangSecurity,

> The ratio is 1% stZVE and 99% vstZVE. The attacker stakes ZVE, so it's now 2% for stZVE, call disitrbute yield, then unstake ZVE, so it's again 1%.

`distributeYield` is only called every 30 days. The yield during these 30 days is distributed based on a snapshot of the total supplies at the time `distributeYield` is called. This issue claims that if you take a flashloan of ZVE before calling `distributeYield` and stake, increasing the stZVE supply, the snapshot is taken, and the % of yield attributed to stZVE for the 30 days is increased. After 30 days, another `distributeYield` call happens and a snapshot is taken again, where the % is back to 1% as the flashloanee would have unstaked the stZVE by then.

My claim is that this is the intended behaviour, the flashloan does not matter here at all because the attacker has to stake to profit from this. If the attacker stakes, then he also rightfully so receives rewards for the increased supply of stZVE.

panprog

@WangSecurity `distributeYield` sets the rate for the next 30 days.

Scenario A: 1% stZVE, 99% vstZVE. `distributeYield` sets the rate for the next 30 days: stZVE get $1/second, vstZVE get $99/second.

Scenario B: 1% stZVE, 99% vstZVE. User acquires 1% ZVE, stakes it, calls `distributeYield`, unstakes, returns back ZVE. The rate set for the next 30 days: stZVE get $2/second, vstZVE get $98/second.

In Scenario B, stZVE stakers get double the rate they should while ZVE is released into circulation as if it was never staked. Basically, it's possible to inflate stZVE reward rate while not really staking corresponding ZVE amount.

0x73696d616f

The design is suboptimal but it is ultimately a design choice. It can not be gamed like this issue says with a flashloan. The supposed attack is just regular user operation. I believe in these cases it is not a valid medium on Sherlock, but I leave it up to the judge.

WangSecurity

I see and understand with arguments both for and against it being the designed decision. But this in fact allows to inflate the stZVE rewards without actually staking ZVE. In the edge case, the attack can be done in one transaction and inflate the rewards in several times (even more than 2x as in the Lead Judge's example). Hence, I believe it's indeed valid, planning to reject the escalation and leave the issue as it is.

RealLTDingZhen

I agree with @0x73696d616f . It should be a design choice here, because this issue cannot be fixed in current protocol. Even if a minimum holding interval is guaranteed, whales can still change the stZVE/vestZVE reward distribution ratio by

Attacker buys and stakes a very large amount of ZVE through a flashloan

> in advance Attacker wait for minimum holding interval Attacker calls ZivoeYDL::distributeYield Attacker collects a very large amount of distributed yields Attacker withdraws and sells back his ZVE tokens effectively stealing undeserved yields

The only way to fix this is to add a withdraw fee, which means a redesign of protocol core logic. So I believe this is not a contract vulnerablilty.

WangSecurity

Thank you for that response! Firstly, I don't think that the mitigation is the reason to validate/invalidate the issue, moreover, the LSW provided two other mitigations. Secondly, I understand that the same can happen with regular workflow, but I believe it's still an issue, cause allows the attacker to intentionally manipulate the distribution ratio without actually staking ZVE (only for one block). Hence, my decision remains the same.

Evert0x

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0x73696d616f: rejected

Issue M-12: OCC_Modular::applyCombine will round APR down

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/304

The protocol has acknowledged this issue.

Found by

cergyk, denzi_, jasonxiale

Summary

`OCC_Modular::applyCombine` calculation of `APR` `APR = APR / notional` rounds in defavor of the protocol, and a user can game this feature to shave of a point of APR from one of his loans.

Vulnerability Detail

`OCC_Modular::applyCombine` is used to combine multiple loans into a single loan.

We can see that the APR for the new loan is computed as a weighted average of combined APRs. However since the division rounds down, the APR can be underestimated by 1 point. Since the APR is expressed as BIPS, a point represents a significant amount of interest.

OCC_Modular.sol#L781

```
APR = APR / notional;
```

> Please note that even in the case where an underwriter has verified off-chain that the APR would not be rounded down before approving a combination, a user can make sure it rounds down by making some payments on his loans

Impact

Loss of funds for the borrowers of the protocol, since a user can reduce APR on his loans by 1 point

Code Snippet

- https://github.com/sherlock-audit/2024-03-zivoe/blob/d4111645b19a1ad3ccc899bea073b6f19be04ccd/zivoe-core-foundry/src/lockers/OCC/OCC_Modular.sol#L781

Tool used

Manual Review

Recommendation

Ensure that there is no rounding error in the `APR` calculation's result by adding a check such as:

```
+ require(APR % notional == 0, "rounding");
APR = APR / notional;
```

Discussion

pseudonaut

Not relevant, also not of concern

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> invalid, the impact is insignificant and impossible to avoid rounding anyway, so it has to happen anyway

CergyK

Escalate

Issue should have the same severity as #416, since it has exactly the same impact (rounding down on a BIPS precision number).

This has the impact of causing a loss of approx 100$ on a 1M$ loan, which is not negligible. A reasonable fix recommendation is provided to avoid rounding down.

sherlock-admin3

> Escalate
>
> Issue should have the same severity as #416, since it has exactly the same impact (rounding down on a BIPS precision number).
>
> This has the impact of causing a loss of approx 100$ on a 1M$ loan, which is not negligible. A reasonable fix recommendation is provided to avoid rounding down.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

panprog

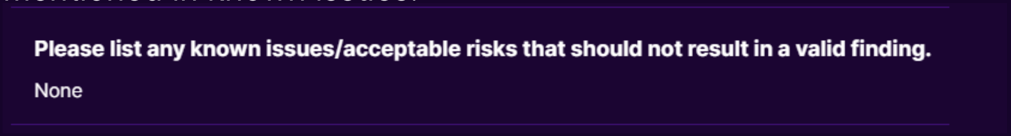
This issue is different from #416, because here it's impossible to avoid rounding due to low precision of APR - so it's either rounded up (user loses funds at the expense of protocol) or rounded down as now (protocol loses funds at the expense of the user). Since it's impossible to avoid funds loss due to APR precision (which is apparently protocol choice), the rounding direction is also a protocol choice. There are no security concerns with either rounding direction here (as opposed to, say, ERC4626, where incorrect rounding direction might cause critical issues in extreme situations).

In #416 the issue is in the order of multiplication and division, which makes it round to the same 0.01% as here, but with the correct order of operations it's possible to avoid this rounding and increase precision. So this is different as it can be fixed and thus can not be considered protocol decision.

omar-ahsan

@panprog with all due respect, every line of code written by devs is a protocol decision on how they want the protocol to act. Watsons have identified a line of code which makes the protocol lose funds. The decision that they have made makes the protocol lose funds, this is a major concern in my opinion.

If it was known to the protocol that rounding down here makes the protocol lose funds and it is acceptable for the protocol to bear this loss then it should have been mentioned in known issues.



Please list any known issues/acceptable risks that should not result in a valid finding.

None

Furthermore rounding up in the favor of the protocol is the right decision, the protocol's task is to offer combinations. It's the borrower's decision to accept it or let it expire, Users who do not want to overpay (in case it is rounded up) can simply keep on paying the interests per each payment as before.

The loss of funds is always going to occur here for any magnitude of loans. I believe this should be judged according to the best interest of the protocol as its simply the case of protocol losing funds which they should not after an action.

WangSecurity

Need clarification on this and this comments. LSW says it's a loss of $100 on a $1M loan, which is only 0.0001%, when the lead judge says it's 0.01% as well as on #416. So what is the correct percentage here?

panprog

@WangSecurity

100/1000000=0.0001 = 0.01% APR is in BIPS, 1 BIPS = 1/10000 (0.01%)

WangSecurity

I agree that loss is quite low (0.01%), but I believe it exceeds small and finite amounts, therefore sufficient for medium severity. Planning to accept the escalation and validate the report with medium severity.

Are there other issues which identified this issue?

marchev

@WangSecurity Unfortunately, I don't think I can escalate one of my reports #671. It is not the same issue however it is an issue that identifies an attack path that results in a loss of 0.17% APR for some loans. Given that a loss of 0.01% warrants a valid issue can I please ask you to take a look at #671?

panprog

@WangSecurity My main point of why it's invalid is because rounding is inevitable and either choice is valid. If this issue is fixed (by rounding up), a new valid issue may be raised that user loses the same amount of funds due to APR rounding up. Why do you think that user must lose this amount, and not protocol (like it is now)? I believe this is the design choice and thus not valid issue.

I can understand that low precision might be the core reason, but this, again, is a design choice to limit APR to 1/10000 precision.

So I disagree because I think this is simply design choice, not because the amount lost is too small.

If you do decide to validate this as medium, here are dups: #681, #684

panprog

To add on - this issue recommends to revert if there is any rounding error at all (only make it work if it divides exactly), but I believe that this is, again, design decision (and in my opinion not the best one - why limit user to only specific percentages?).

Since combinations are approved by admin, he should simply not approve combines which might lead to bad combined loan APR. So if the recommendation of this issue is implemented, then this means that this issue is simply admin mistake who approves the loan combine he it trusted to not approve.

CergyK

> To add on - this issue recommends to revert if there is any rounding error at all (only make it work if it divides exactly), but I believe that this is, again, design decision (and in my opinion not the best one - why limit user to only specific percentages?).

> Since combinations are approved by admin, he should simply not approve combines which might lead to bad combined loan APR. So if the recommendation of this issue is implemented, then this means that this issue is simply admin mistake who approves the loan combine he it trusted to not approve.

That's an interesting point, however please notice that even if underwriter approves a combine which does not lead to precision loss, a user can still put it into rounding down state by making a repayment on a loan

omar-ahsan

> If this issue is fixed (by rounding up), a new valid issue may be raised that user loses the same amount of funds due to APR rounding up. Why do you think that user must lose this amount, and not protocol (like it is now)? I believe this is the design choice and thus not valid issue.

The new valid issue which has been mentioned can be considered design decision as the protocol should not be losing funds via interests which they were receiving before combination is applied. Further more in situations like these, rounding up is considered in favor of the protocol.

The protocol should not be taking accruing losses per every approved combination, Consider 100 approved combinations and 100 borrowers accepts them, the protocol is now taking a loss per every combination, in the case of user, it is one single user overpaying a small amount per their combination. Borrowers will always accept the combination if they know they will have to pay less. The protocol suffers more here than each single user so this issue should be judged in the best interest of the protocol.

Its a two step process, The underwriter has to approve and then the user has to accept, If the APR is rounded up and if the borrower does not want to overpay in exchange of combining their loan then they can let the combination expire and keep on paying in separate transactions.

WangSecurity

Very insightful comments and I think I confused you a bit with my wording. Firstly, I don't think the mitigation should effect the validity of the report. Secondly, great point about admin mistake, but as LSW said here can make it round down. But I actually would like to clarify if it can be done intentionally and asking to give a small numeric example? Thirdly, I agree that it's design to make such low precision, but if it can be done intentionally, leading to other users lose funds. Hence, I may reconsider my decision after getting the answer if it can be done intentionally.

CergyK

> Very insightful comments and I think I confused you a bit with my wording. Firstly, I don't think the mitigation should effect the validity of

> the report. Secondly, great point about admin mistake, but as LSW said here can make it round down. But I actually would like to clarify if it can be done intentionally and asking to give a small numeric example? Thirdly, I agree that it's design to make such low precision, but if it can be done intentionally, leading to other users lose funds. Hence, I may reconsider my decision after getting the answer if it can be done intentionally.

Yes a user can make a few repayments intentionally to make the combine round down again. The case in which the combine does not round down is extremely specific, so any small deviation of the values will lead it to round down again

Let's take an example:

Loan A: 100 repayments left 100 principal 20 APR

Loan B: 100 repayments left 100 principal 50 APR

a combination of these loans would yield an APR: (50x100+20x100)/(100+100)=35

without losing precision, because the division is exact

However once the combine is approved, the user can repay one payment on loan B (always can do this action):

Loan B: 99 repayments left 99 principal 50 APR

now the combination yields: (50x99+20x100)/(100+99) = 34

Whereas the parameters of the loans have barely changed

WangSecurity

Therefore, I agree that this is the design decision to have small precision, but with the fact that it allows a user to intentionally make APR round down causing a loss of funds. Hence, I believe it's suitable for medium severity. Planning to accept the escalation and validate the report with medium severity.

panprog

Agree that as the user can influence this rounding in his favor, then this is a valid medium.

Evert0x

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- CergyK: accepted

omar-ahsan

@Evert0x I think you have mistakenly set this as unique, The lead judge has identified #681 and #684 as duplicates in his comment here

Issue M-13: `ZivoeYDL::earningsTrancheuse()` always assumes that `daysBetweenDistributions` have passed, which might not be the case

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/347

The protocol has acknowledged this issue.

Found by

0x73696d616f

Summary

`ZivoeYDL::earningsTrancheuse()` calculates the target yield, senior and junior proportions based on `daysBetweenDistributions`. However, it is not strictly enforced that `ZivoeYDL::distributeYield()` is called exactly at the `daysBetweenDistributions` mark, leading to less yield for tranches.

Vulnerability Detail

There is a yield target for senior and junior tranches over a daysBetweenDistributions period that when exceeded, sends the rewards to residuals. However, the current code will not strictly follow the expected `APY` for senior tranches, as the earning of tranches are calculated based on `daysBetweenDistributions`, but `ZivoeYDL::distributeYield()` may be called significantly after the `daysBetweenDistributions` mark.

Add the following test to `Test_ZivoeYDL.sol` to confirm that when the senior tranche yield reached the expected target for `30` days, if the time that `ZivoeYDL::distributeYield()` was called increased, the yield distributed to the senior tranch remains the same.

```
function test_POC_ZivoeYDL_distributeYield_notEnforcingTranchesPeriod() public {
    uint256 amtSenior = 1500 ether; // Minimum amount $1,000 USD for each coin.
    uint256 amount = 1000 ether;

    // Simulating the ITO will "unlock" the YDL
    simulateITO_byTranche_optionalStake(amtSenior, true);

    // uncomment this line instead to confirm that it's the same value below
    //hevm.warp(YDL.lastDistribution() + YDL.daysBetweenDistributions() * 86400);
    hevm.warp(YDL.lastDistribution() + YDL.daysBetweenDistributions() * 86400 +
 ↪  1 days);
```

```
    // Deal DAI to the YDL
    mint("DAI", address(YDL), amount);

    YDL.distributeYield();

    console.log(IERC20(DAI).balanceOf(address(stSTT)));
}
```

Impact

Guaranteed less `APY` for junior and senior tranches. The amount depends on how much time the call to `ZivoeYDL::distributeYield()` was delayed since `block.timestamp` reached `lastDistribution + daysBetweenDistributions * 86400`.

Code Snippet

distributeYield()

```
function distributeYield() external nonReentrant {
    ...
    require(
        block.timestamp >= lastDistribution + daysBetweenDistributions * 86400,
        "ZivoeYDL::distributeYield() block.timestamp < lastDistribution +
↪   daysBetweenDistributions * 86400"
    );
    ...
```

ZivoeYDL::earningsTrancheuse()

```
function earningsTrancheuse(uint256 yP, uint256 yD) public view returns (
    uint256[] memory protocol, uint256 senior, uint256 junior, uint256[] memory
↪   residual
) {
    ...
    uint256 _seniorProportion = MATH.seniorProportion(
        IZivoeGlobals_YDL(GBL).standardize(yD, distributedAsset),
        MATH.yieldTarget(emaSTT, emaJTT, targetAPYBIPS, targetRatioBIPS,
↪   daysBetweenDistributions),
        emaSTT, emaJTT, targetAPYBIPS, targetRatioBIPS, daysBetweenDistributions
    );
    ...
}
```

Tool used

Manual Review

Vscode

Foundry

Recommendation

Modify the ZivoeMath::seniorProportion() and ZivoeMath::yieldTarget() implementations to use seconds instead of days for `T`. Then, in `Zivoe::YDL::earningsTrancheuse()`, instead of daysBetweenDistributions when calculating the yield target and senior proportion, use `block.timestamp - lastDistribution`. Also, in `ZivoeYDL::distributeYield()`, `lastDistribution = block.timestamp;` must be moved after earningsTrancheuse() is called.

Discussion

pseudonaut

It will called immediately when possible, so there may be few blocks of lag, acceptable

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> invalid, trusted keepers will call it as soon as possible, and even there is a slight delay due to network congestion or whatever else, this is acceptable and intended

0x73696d616f

Escalate

This should be valid as in the first distribution period, the duration is 60 instead of 30, calculating a much smaller APR than it should be. Half the APR is calculated instead of the intended, a substancial amount, which means this should be a valid medium according to the docs.

sherlock-admin3

> Escalate
>
> This should be valid as in the first distribution period, the duration is 60 instead of 30, calculating a much smaller APR than it should be. Half the APR is calculated instead of the intended, a substancial amount, which means this should be a valid medium according to the docs.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

panprog

This is still invalid/low. The 30-days yield for the first distribution which is after 60 days is also intended according to sponsor, see for example #531 which is about the same.

0x73696d616f

Where did the sponsor state or are there any docs showing that the APR should also be half? The sponsor only mentioned the duration as intended.

panprog

This was in a private discussion to confirm the intended behavior.

0x73696d616f

Can't argue with that. In any case, just a small delay of 5 minutes, will lead to more than 1 BIPS of loss of rewards, as the duration of 30 days is relatively small. `(5 / (30*24*60))*10000 = 1.15`. The delay may be bigger in certain sporadic events, leading to even bigger losses.

0x73696d616f

Additionally, the sponsor may say it's acceptable but it does not mean it is not a significant amount nor invalidates the issue.

0x73696d616f

This article, for example, mentions an outage for 1 hour, which is `(1 / (30*24))*10000 = 14 BPS`. Much more significant than dust or precision, as the docs mention to be a valid medium issue.

WangSecurity

Firstly, I agree that there is no guarantee the keeper will make the call instantly, and there might be delay. But, for that I need to understand what are the losses. I see in this comment you say that in 5 mins it will lose 1 BIPS in terms of rewards, which is 0.01% of the rewards lost, correct? And if it's less than 5 mins, then it'll be less than 1 BIPS. But, the longer the delay, the larger the loss, correct?

0x73696d616f

@WangSecurity exactly.

panprog

@WangSecurity Example:

- Tranche has $3600 deposited. Target APR = 10%, meaning it's $30 per month.
- The earnings for previous period = $35. $30 is deposited to tranche ZivoeReward and distributes at a rate of $1/day (the rest of it - $5 - goes to residual recepients)
- The earnings for the next period = $32. However, the keeper failed to call it in time and called only after 1 day. Since the distribution was $1/day for 30 days, it stops after 30 days and it's $0 for the following day.
- The keeper finally calls `distributeYield` after 1 day of delay, but it distributes $30 again (not $31 to account for extra day), $2 goes to residuals.

This means that due to 1 day of keeper delay, tranche stakers will receive $30 over 31 days instead of $31. This is 1/31 = 3.2% loss. However, perhaps a more correct way to calculate is to compare the target APR with actual APR. So target = 10%, actual = 30/31*356/3600 = 9.57%, a loss of 0.43% APR per day of delay or a loss of 0.0003% APR per minute of delay.

Since such keeper delay is not very likely, it's better to estimate a total time of keeper delay over the year. In such case the APR loss is 1/3600 = 0.028% per day of delay. Again, this is only if distributed yield exceeds the target APR.

One more thing to consider is that the target rates can be changed by admin via `updateTargetAPYBIPS` and `updateTargetRatioBIPS`, so admin can compensate for these delays by increasing target rate if necessary.

CergyK

Additional points to take into consideration when evaluating this issue:

- `distributeYield` is an unpermissioned action, so if the impact of keeper downtime is deemed unacceptable by any actor, they can ensure it is called timely themselves.
- A mainnet outage event of which 2 happened last year, would have to coincide exactly with the beginning of a `distributeYield` unlocking, which is quite unlikely (1 hour / 30 days = 0.01% chance of overlap)

0x73696d616f

@panprog your comment is highly misleading. Firstly, you make a logic error (I hope it's a mistake), which will be explained in detail. Then, replace the `3600` by `360_000`, and now the loss is 100 USD. Also, the previous and next periods are not required for this discussion, only the current distribution. There is no need to add extra complexity. Let's break it down.

The expected monthly APR is pro-rata to the time passed, which means the error % will be the same. So a 3.2% error in the number of days will lead to a 3.2% APR error during the 31 days. Check the math at the end of the comment for confirmation. This is the first mistake.

Now, the second mistake, which is much more serious, is that @panprog is calculating the error in ABSOLUTE terms, not relative. This makes no sense. For example, if the target APR was `0.1%`, the actual APR would be `0.001 - 0.001*0.032 = 0.000968`, just a difference of `0.0032%`. This is obviously wrong and extremely misleading. The actual error is calculated in relative terms, similar to what is done when calculating the days error as 1/31, and not just 1. So, if the target APR is 10%, but the actual is 9.68%, the error in % is (target - actual) / target = `(0.1 - 0.0968) / 0.1 = 0.032`, or 3.2%.

Additionally, it makes no sense to compare to the yearly APR because e would have to sum all the delays in the year to calculate it, not just use the delay of a single month.

Math proof that the APR error is always the same as the number of days error: Target USD for 31 days, USDt, is `USDt = 31 / 365 * 3600 * 0.1`. 31 is the number of days, 365 the number of days in a year, used in the calculation and 0.1 is 10%. Now, the actual % in these 31 days, where USDa will be the actual USD in the 31 days: `USDa = 3600 * 0.1 * 30 / 365`. This is because during the 31 days, only 30 is distributed. To get the corresponding %, we invert the formula above, where 0.1 is the target % (10%) `0.1 = USDt * 365 / 31 / 3600`, which is just the inverted formula above. Now, to get the actual %, just replace the right hand side of the expression before by the actual values `APR = USDa * 365 / 31 / 3600`, `USDa = 3600 * 0.1 * 30 / 365`, so `APR = 3600 * 0.1 * 30 / 365 * 365 / 31 / 3600 = 0.1*30/31`, which is 0.0968. Sounds familiar right? this is 0.1 - 0.0032, where 0.0032 is the 3.2% calculated before, but multiplied by 0.1 because it's in APR terms.

0x73696d616f

Other points to consider:

1. If there is an outage it does not matter who is the actor calling distribute yield.
2. The chance is small for a big outage, but smaller to medium delays are much more likely (where smaller delays are guaranteed).

WangSecurity

Firstly, as I understand all of us agree that the vulnerability is present in the code. The problem here is that there is firstly, high constraints:

1. Keepers not calling distribute yield.
2. Users not calling distribute yield as well.

But there is no guarantee that the keeper will cetainly make a call and not misbehave and there is no guarantee that the user will call it on time. So I believe, it's fair to say that the daly may indeed happen, but with very low probability. To better understand what are the losses, @0x73696d616f can you make these calculations if it's only 1 hr delay or 5 min delay? And @panprog and @CergyK would like to ask you check the calculations above to verify they're 100% correct.

0x73696d616f

The calculations are very simple, the loss is delay / (30 days + delay) × earnings. One hour is 1 / (30x24 + 1) = 0.13%. 5 minutes is 5 / (30x24x60 + 5) = 0.01%. If the earnings are 360_000, the loss is 500 and 42 USD, respectively.

CergyK

> The calculations are very simple, the loss is delay / (30 days + delay) × earnings. One hour is 1 / (30x24 + 1) = 0.13%. 5 minutes is 5 / (30x24x60 + 5) = 0.01%. If the earnings are 360_000, the loss is 500 and 42 USD, respectively.

Agree with these calculations

panprog

> The calculations are very simple, the loss is delay / (30 days + delay) × earnings. One hour is 1 / (30x24 + 1) = 0.13%. 5 minutes is 5 / (30x24x60 + 5) = 0.01%. If the earnings are 360_000, the loss is 500 and 42 USD, respectively.

Yes, the calculations are correct.

WangSecurity

As I understand, the admin relies on keeper bots to call `distributeYield`, but as we agree there can be a delay and it's not guaranteed that they will call on time every time. Or am I missing something here?

0x73696d616f

@WangSecurity yes, that's it. I think the previous comment ignored a good chunk of this conversation.

ZdravkoHr

If this issue is deemed valid, #85 should also be validated, because:

- it has the same impact, but on a bigger scale - the delay is 30 days and it's certain.
- it does not rely on keeper being inactive - will happen for sure

0x73696d616f

@ZdravkoHr I don't think they should be grouped together, you'll notice I submitted a separate issue #345 which was exactly like #85 and was invalidated for the same reason (sponsor's intention for the first distribution to be 60 days). In #345 and

#85, they both point at the fact that the initial distribution is 60 days, not 30. This issue goes a bit further and claims that beyond the initial period, there will be more delays as the keeper can not guarantee strictly calling it on time. This is further demonstrated by the recommendations given, using the one from #85, this issue (#347) would still exist. Thus, #85 does not identify the correct attack path.

ZdravkoHr

@0x73696d616f, I agree #85 doesn't talk about `daysBetweenDistributions` in more detail, but the root issue is still the same, as described

```
Therefore, the first distribution can be executed after at least 60 days since
↪  the unlocking have passed. However, the whole distribution process
↪  calculates its values for a single month, using daysBetweenDistributions.
```

If `earningsTrancheuse` was not fixated to 30 days, but instead it calculated time correctly, there would have been no problem with initial distribution happening after 60 days.

0x73696d616f

@ZdravkoHr I know you were aware of the issue when you wrote the report (as you correctly identified the root cause), but the Sherlock rules are clear, if the identified attack path is incorrect (as is the case in #85 as your whole report only mentioned the initial 60 days period), the issue is at most low.

ZdravkoHr

```
Both B & C would not have been possible if error A did not exist in the first
↪  place. In this case, both B & C should be put together as duplicates.

In addition to this, there is a submission D which identifies the core issue but
↪  does not clearly describe the impact or an attack path. Then D is considered
↪  low.
```

I don't think #85 is `issue D` in this scenario. There isn't really an attack path in this situation. There is a root cause which causes loss in yield based on a given delay. The delay in #85 is caused by executing the initial distribution 30 days later, while the delay in this issue is caused by a keeper being inactive.

WangSecurity

As we agree, the keeper might have a delay and not guaranteed to call on time. Secondly, despite it's being in the user interest to call it on time as well, it's not guaranteed as well. Thirdly, as was proven by the escalator the loss is equal or exceeds 0.01% which I believe exceeds small and finite amounts. Finally, as I understand, the sponsor said it's acceptable only after the contest and it wasn't known during the contest that this risk is known/acceptable.

Hence, planning to accept the escalation and validate the issue with medium severity.

ZdravkoHr

@WangSecurity, what do you think about #85?

0x73696d616f

#85 explicitly says the root cause is the initial period being 60 days. It mentions that the yield is calculated on 30 days, but does not see this part as the root cause.

RealLTDingZhen

Hey @WangSecurity @panprog after a careful reading, I believe this issue is valid, and should be duplicated with #697 #692 #672 #657.

Zivoe didn't tell watsons whether they were going to run a keeper themselves or use an external keeper. Lets assume for this issue users will not calling distribute yield.

If protocol is going to run a keeper themselves, then this is not a issue because admin/owner is TRUSTED.

If the external Keeper does not follow the rules for calling the `distributeYield()` on time, this is a misbehave of external admin, so this issue should be duplicated with those where external admin's actions are leading to a harm of Zivoe with Medium severity.

> Are the admins of the protocols your contracts integrate with (if any) TRUSTED or RESTRICTED?
>
> RESTRICTED

And please refer to here:

https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/672#issuecomment-2115546327

> according to README and external admins being restricted, these reports indeed should be valid. But, I believe the most fair option is to duplicate this report with other issues where external admin's actions are leading to a harm of Zivoe with Medium severity, due to extremely low likelihood.

0x73696d616f

> Lets assume for this issue users will not calling distribute yield.

This assumption makes no sense.

> If protocol is going to run a keeper themselves, then this is not a issue because admin/owner is TRUSTED.

This is false as they can not control network congestion.

RealLTDingZhen

If this assumption makes no sense, this issue should be invalidated If users will call distribute yield immediately, this is not a issue because sponsor confirms

> It will called immediately when possible, so there may be few blocks of lag, acceptable

Please refer to https://docs.sherlock.xyz/audits/judging/judging#vii.-list-of-issue-categories-that-are-not-considered-valid

> #20: Chain re-org and network liveness related issues are not considered valid. Exception: If an issue concerns any kind of a network admin (e.g. a sequencer), can be remedied by a smart contract modification, the protocol team considers external admins restricted and the considered network was explicitly mentioned in the contest README, it may be a valid medium. It should be assumed that any such network issues will be resolved within 7 days, if that may be possible.

This issue is not a vulnerability if **one of** keepers or protocol team or users is behave normally. In your words, it is a issue because of network congestion, which is not accepted by sherlock.

The only possible path for this issue is a external Keeper bot misbehave, which should be duped with those issues above.

0x73696d616f

> If this assumption makes no sense, this issue should be invalidated If users will call distribute yield immediately, this is not a issue because sponsor confirms

network congestion.

A small delay is not considered 'network liveness', it's regular behaviour. And the issue may be fixed by a smart contract modification.

RealLTDingZhen

As you agree there may be a small delay:

https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/347#issuecomment-2081539634

> there may be few blocks of lag, acceptable

So, what kind of network congestion(not a issue with network liveness) can make this a issue? If this congestion occurs on the Keeper side,

Per LSW:https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/347#issuecomment-2114623235

> A mainnet outage event of which 2 happened last year, would have to coincide exactly with the beginning of a distributeYield unlocking, which is quite unlikely (1 hour / 30 days = 0.01% chance of overlap)

This is definitly low likelihood + low impact(keepers should call the function right after the congestion, so only a few blocks of lag)

0x73696d616f

@RealLTDingZhen the numbers are all above, please refer to them.

RealLTDingZhen

I see your calculations ser, but this did not counter my(and Cergyk) point.

> the loss is delay / (30 days + delay) × earnings

I agree that the losses will increase in that proportion, but this do not counter my arguments.

My two main arguments are:

If we assume this issue do not require a malicious Keeper and protocol team/users are all not aware of this, then:

1. the likelihood is still extremely low. The "network congestion" have to coincide exactly with the beginning of a distributeYield unlocking. (@CergyK can you please confirm how long the 2 mainnet outage event in last year last?)
2. This issue relays on a long Ethereum mainnet downtime. As you have calculated, 1 hour lag = 0.13% loss of yield that month. However, an hour of downtime on Ethereum mainnet can definitely be seen as a network liveness issue. (Consider those lending/perpetual/oracle protocols, it would be disastrous for them.) If the lag is just some blocks, sponsor is fully acknowledged and can accept the risk, so this is a informational issue.

0x73696d616f

The likelihood goes from very high to low as the duration of the delay goes up. You're picking on the upper bound and basing the argument on it, which is deceiving.

RealLTDingZhen

If the network is congested for a long period of time, then this issue can be invalidated outright by reason "network liveness". If the network is congested for a short period of time, then this issue can be invalidated outright because sponsor confirms there may be few blocks of lag, and that is acceptable.

So how long is the right delay to make this issue both probable and impactful? And my points above is still valid to invalidate this issue.

0x73696d616f

There is no clear line as to where the delay stops being considered acceptable. It's up to the user deciding the escalations to decide, which has been done.

> this issue can be invalidated outright because sponsor confirms there may be few blocks of lag, and that is acceptable.

This has also been answered.

RealLTDingZhen

This is my final statement on this issue (Summary of above views):

1. If we think this issue need a malicious/misbehave Keeper bot, then this should be duped with #697 #692 #672 #657.
2. If we think this issue can be triggered by normal operation, then this issue should be invalidated for reasons below: a. The likelihood is extremely low: A mainnet outage event of which 2 happened last year, would have to coincide exactly with the beginning of a distributeYield unlocking, which is quite unlikely. And even if such coincidence downtime exists, the issue becomes a network liveness issue because the loss is proportional to the downtime(5min -> 1BIPS, which is a network liveness issue.). b. The impact is low: 5min downtime in Ethereum-> 1 BIPS loss itself is low impact. and sponsor confirms that "there may be few blocks of lag, acceptable".

RealLTDingZhen

Let's leave it up to the judge for now, it's ultimately up to him. Good discussion @0x73696d616f I always like a good debate

0x73696d616f

> If we think this issue can be triggered by normal operation, then this issue should be invalidated for reasons below: a. The likelihood is extremely low: A mainnet outage event of which 2 happened last year, would have to coincide exactly with the beginning of a distributeYield unlocking, which is quite unlikely. And even if such coincidence downtime exists, the issue becomes a network liveness issue. b. The impact is low: 5min downtime in Ethereum downtime -> 1 BIPS loss itself is low impact. and sponsor confirms that "there may be few blocks of lag, acceptable".

Your message is incorrectly implicitly mixing the likelihood of the 1 hour event with the impact of a 5 minutes delay, Also, please don't mention point 1. as this was never mentioned in this report, you're just adding confusion.

RealLTDingZhen

> Your message is incorrectly implicitly mixing the likelihood of the 1 hour event with the impact of a 5 minutes delay, Also, please don't mention point 1. as this was never mentioned in this report, you're just adding confusion.

This is because your issue requires distribute yield function not be called for a period of time, and there are only two scenarios for this - Keeper misbehave or network downtime.

WangSecurity

I hope this will answer your questions:

Firstly, there is no guarantee the keeper will work without a delay. Secondly, if we consider it external, I believe it's not a dup of #657 because 657 requires external admins executing certain actions to completely rug Zivoe (either block their functionality or steal funds), while this can happen at any time, cause it's not guaranteed to execute every call on time every time. Thirdly, I see the sponsor saying it's acceptable, but it's said only after the contest and wasn't known during it. Fourthly, I agree that the loss is very small, and the likelihood as well. But it still qualifies for medium severity: high external constraints and the loss of funds exceeds small and finita amounts. I believe 0.01% exceeds small and finite amounts. If any of my assupmtions is wrong, please correct me or ask questions.

But the decision remains the same, accept the escalation and validate the report with medium severity.

ZdravkoHr

@0x73696d616f, I agree in my report I have described setting the lastDistributed as a root cause. But I do still think that #85 deserves revisiting, even if it's not judged as a duplicate of this issue because the impact is the same if not worse

Evert0x

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0x73696d616f: accepted

Issue M-14: Title: Inadequate Allowance Handling in convertAndForward Function of `OCT_DAO` & `OCT_YDL`.

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/609

Found by

0xBhumii, recursiveEth

Summary

The `OCT_DAO:convertAndForward` and `OCT_YDL:convertAndForward` function suffers from inadequate handling of token allowances for the 1inch router. Although allowances are set correctly before converting assets, they are not reset afterward. This can lead to failed transactions if the 1inch router does not utilize the entire allowance due to slippage or other factors.

Vulnerability Detail

the issue arises from the lack of allowance reset after interacting with the 1inch router. If the router does not utilize the entire allowance specified due to slippage or other reasons, the allowance will remain unchanged, potentially causing the assertion to fail and the transaction to revert.

Impact

The impact of this vulnerability is that transactions may fail due to incorrect allowance management. This could result in inefficiencies in liquidity provision, loss of gas fees because due to asset statement it consume all the gas and won't return anything and convertandTransfer will never be able to transfer asset to DAO.

Code Snippet

https://github.com/sherlock-audit/2024-03-zivoe/blob/main/zivoe-core-foundry/src/lockers/OCT/OCT_DAO.sol#L87 https://github.com/sherlock-audit/2024-03-zivoe/blob/main/zivoe-core-foundry/src/lockers/OCT/OCT_YDL.sol#L97

```
assert(IERC20(asset).allowance(address(this), router1INCH_V5) == 0);
```

Tool used

Manual Review

Recommendation

Reset Allowances After Use: After interacting with the Uniswap router and completing the liquidity provision, reset the allowances for the pair assets and ZVE tokens to zero to ensure they are not left with excessive allowances.

Discussion

pseudonaut

Valid

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> borderline low/medium, the report has just generic suggestion without specific POC (exact scenario) when this can actually happen. Normally, when doing a swap, full allowance is used to do it. It appears that 1inch's fillOrderRFQ might not use full allowance if there is no matching order in the orderbook, but I'm not 100% sure. Watson should have provided the POC to prove his point.

BiasedMerc

This seems like it should be invalid, as mentioned by lead judge, there is no POC or explanation as to how any allowance can be left after interaction with the router, and all explanations and suggestions are generic.

I feel like for this type of issue some sort of written POC should be required to prove the issue, as currently there is no way to prove if this scenario can actually happen. And judge also is unsure if this is really a valid issue from the comment, so it seems the warden has not done an adequate job at proving the issue due to lack of POC.

E.g. #18 describes a similar issue for UniSwap and outlines the exact scenario of how leftover allowance can occur with code-snippets.

IronsideSec

In the vulnerability section, watson says `If the router does not utilize the entire allowance specified due to slippage or other reasons` submit a POC to prove that 1inch will operate in a way that allowance will be > 0 in any of the success swap paths, then issue might be valid.

swaps will revert if the slippage did not match. If swap fails, whole transaction will revert, hence chances for allownce to be > 0 after a successful swap is not possible

Afriaudit

As far as I understand here. There was a flaw in the protocol's implementation when engaging with a third party.(in this case uniswap, curve, sushi, 1inch) and the flaw is: -give allowance to the third party -Third party withdraws from the protocol -Protocol fails to reduce allowance left -Protocol proceeds to assert allowance = 0

This implementation will cause recurring reversion when interacting these third parties.

According to sherlock rules; "In case the same vulnerability appears across multiple places in different contracts, they can be considered duplicates. The exception to this would be if underlying code implementations, impact, and the fixes are different, then they can be treated separately." As far as I know the impact and fixes are same making It a dup of #18. Though the implementation logic is same the exact code Is different, one being uniswap the other being one inch. It will be Interesting to see how this turns out as it will help understand the interpretation of the above sherlock rule.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Zivoe/zivoe-core-foundry/pull/262

armormadeofwoe

There are currently 2 separate issues regarding vulnerable `assert` statements in swaps: #18 with UniSwap and this one for 1inch. Since the root cause is the same (flawed dev design choice of using `assert`) why are they considered as separate bugs?

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-15: DAO unable to withdraw their funds due to Convex admin action

Source: https://github.com/sherlock-audit/2024-03-zivoe-judging/issues/657

The protocol has acknowledged this issue.

Found by

AllTooWell, Bauchibred, BiasedMerc, BoRonGod, KingNFT, ZanyBonzy, rbserver, saidam017

Summary

Convex admin action can lead to the fund of Zivoe protocol and its users being stuck, resulting in DAO being unable to push/pull assets from convex_lockers.

Vulnerability Detail

Per the contest page, the admins of the protocols that Zivoe integrates with are considered "RESTRICTED". This means that any issue related to Convex's admin action that could negatively affect Zivoe protocol/users will be considered valid in this audit contest.

> Q: Are the admins of the protocols your contracts integrate with (if any) TRUSTED or RESTRICTED?
>
> RESTRICTED

In current `BaseRewardPool.sol` used by convex, admin can add infinite `extraRewards`:

```
function extraRewardsLength() external view returns (uint256) {
    return extraRewards.length;
}

function addExtraReward(address _reward) external returns(bool){
    require(msg.sender == rewardManager, "!authorized");
    require(_reward != address(0),"!reward setting");

    extraRewards.push(_reward);
    return true;
}
```

By setting a malicious token or add a lot of tokens, it is easy to completely forbid Zivoe DAO to `pullFromLocker`, since claimRewards() is forced to call:

```
function pullFromLocker(address asset, bytes calldata data) external override
↪   onlyOwner {
    require(asset == convexPoolToken, "OCY_Convex_C::pullFromLocker() asset !=
     ↪   convexPoolToken");

    claimRewards(false);
    ...
```

Impact

The fund of Zivoe protocol and its users will be stuck, resulting in users being unable to withdraw their assets.

Code Snippet

https://github.com/convex-eth/platform/blob/main/contracts/contracts/BaseRewardPool.sol#L109

Tool used

Manual Review

Recommendation

Ensure that the protocol team and its users are aware of the risks of such an event and develop a contingency plan to manage it.

Discussion

pseudonaut

I suppose valid, not of concern though

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

panprog commented:

> invalid, while admins are restricted, they are still not supposed to harm their protocol, just do normal reasonable admin actions. Adding so many rewards tokens will definitely break their protocol for all users, so this is invalid assumption.

RealLTDingZhen

escalate

This is a valid medium per external admin is restricted and sponsor confirms this.

Please refer to https://github.com/sherlock-audit/2024-01-napier-judging/issues/95 and https://github.com/sherlock-audit/2024-01-napier-judging/issues/108

sherlock-admin3

> escalate
>
> This is a valid medium per external admin is restricted and sponsor confirms this.
>
> Please refer to https://github.com/sherlock-audit/2024-01-napier-judging/issues/95 and https://github.com/sherlock-audit/2024-01-napier-judging/issues/108

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

panprog

Even if this happens, there is nothing that can be done about it, so this is purely informational issue. It won't affect the other protocol functionality since this is just a separate locker with funds locked in it. I believe this is similar to USDC blacklisting protocol address, so such actions should be invalid as there is nothing that can be done to fix it.

WangSecurity

I agree with both the Lead Judge and escalating Watson, but according to README and external admins being restricted, these reports indeed should be valid. But, I believe the most fair option is to duplicate this report with other issues where external admin's actions are leading to a harm of Zivoe with Medium severity, due to extremely low likelihood.

The reports it will be duplicated with are #672, #692 and #697. Planning to accept the escalation and duplicate these reports.

Evert0x

Result: Medium Has Duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- RealLTDingZhen: accepted

panprog

The following issues appear to be dups to this one according to the @WangSecurity criteria of "external admin's actions are leading to a harm of Zivoe": #130, #160, #249, #648, #662, #666, #667, #672, #677, #692, #697, #699, #701, #704

Disclaimers

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

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