SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Fair Funding
Prepared by:	Sherlock
Lead Security Expert:	hickuphh3
Dates Audited:	February 14 - February 17, 2023
Prepared on:	March 8, 2023

Introduction

Fair Funding allows you to invest in early stage projects while limiting your downside risk. You will get the benefits of early investors yet be sure to get your invest back thanks to Alchemix.

Scope

- `fair-funding/contracts/AuctionHouse.vy`
- `fair-funding/contracts/Vault.vy`
- `fair-funding/contracts/solidity/MintableERC721.sol`

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
3	2

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

hickuphh3
jkoppel
0x52
oxcm
Bauer
ABA
rvierdiiev
csanuragjain
weeeh_
0xSmartContract
minhtrng
Ruhum
0xlmanini
ck
XKET
seyni
carrot
0xhacksmithh
7siech
HonorLt
Bahurum

Issue H-1: amount_claimable_per_share accounting is broken and will result in vault insolvency

Source: https://github.com/sherlock-audit/2023-02-fair-funding-judging/issues/44

Found by

0x52, jkoppel, oxcm

Summary

Claim accounting is incorrect if there is a deposit when amount_claimable_per_share !=0, because position.amount_claimed isn't initialized when the deposit is created.

Vulnerability Detail

https://github.com/sherlock-audit/2023-02-fair-funding/blob/main/fair-funding/contracts/Vault.vy#L430-L440

When calculating the amount of WETH to claim for a user, the contract simply multiplies the share count by the current amount_claimable_per_share and then subtracts the amount that has already been paid out to that token holder. This is problematic for deposits that happen when amount_claimable_per_share != 0 because they will be eligible to claim WETH immediately as if they had been part of the vault since amount_claimable_per_share != 0.

Example; User A deposits 1 ETH and receives 1 share. Over time the loan pays itself back and claim is able to withdraw 0.1 WETH. This causes amount_claimable_per_share = 0.1. Now User B deposits 1 ETH and receives 1 share. They can immediately call claim which yields them 0.1 WETH (1 * 0.1 - 0). This causes the contract to over-commit the amount of WETH to payout since it now owes a total of 0.2 WETH (0.1 ETH to each depositor) but only has 0.1 WETH.

Impact

Contract will become insolvent

Code Snippet

https://github.com/sherlock-audit/2023-02-fair-funding/blob/main/fair-funding/contracts/Vault.vy#L200-L232

Tool used

Manual Review

Recommendation

position.amountClaimed needs to be initialized when a deposit is made:

```
    # deposit WETH to Alchemix
    shares_issued: uint256 = self._deposit_to_alchemist(_amount)
    position.shares_owned += shares_issued
+   position.amount_claimed += shares_issued * amount_claimable_per_share
    self.total_shares += shares_issued

    self.positions[_token_id] = position
```

Discussion

HickupHH3

Dup of #114

0x00052

Escalate for 1 USDC

Not a dupe of #114. That focuses on depositing twice to the same token (which can't happen outside of admin abuse). The issue is that it generally applies to all deposits. Please re-read my example as to why this is an issue and how it leads to gross over-commitment of rewards.

Two reasons I disagree with this being low:

1) The argument that this will only be used for a short period and so "it doesn't have much impact" is a poor argument. This is meant as a general utility that anyone can use and they should be able to make a funding period as long as they want
2) This is a serious issue that will lead to rewards being over-committed and the vault WILL go insolvent as a result. The extra fees being paid will be taken from other users and will GUARANTEED cause loss of funds to other users.

Would like to add that this and #113 are the same issue and escalations should be resolved together.

sherlock-admin

> Escalate for 1 USDC
>
> Not a dupe of #114. That focuses on depositing twice to the same token (which can't happen outside of admin abuse). The issue is that it generally applies to all deposits. Please re-read my example as to why this is an issue and how it leads to gross over-commitment of rewards.
>
> Two reasons I disagree with this being low:

> 1) The argument that this will only be used for a short period and so "it doesn't have much impact" is a poor argument. This is meant as a general utility that anyone can use and they should be able to make a funding period as long as they want
>
> 2) This is a serious issue that will lead to rewards being over-committed and the vault WILL go insolvent as a result. The extra fees being paid will be taken from other users and will GUARANTEED cause loss of funds to other users.
>
> Would like to add that this and #113 are the same issue and escalations should be resolved together.

You've created a valid escalation for 1 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

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

Unstoppable-DeFi

Agree, duplicate of #113 and will fix.

hrishibhat

Escalation accepted

Considering this & its duplicate #113 as valid issues

sherlock-admin

> Escalation accepted
>
> Considering this & its duplicate #113 as valid issues

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/fair-funding/pull/8

Issue H-2: Incorrect shares accounting cause liquidations to fail in some cases

Source: https://github.com/sherlock-audit/2023-02-fair-funding-judging/issues/38

Found by

0x52, Bauer, hickuphh3, jkoppel

Summary

Accounting mismatch when marking claimable yield against the vault's shares may cause failing liquidations.

Vulnerability Detail

`withdraw_underlying_to_claim()` distributes `_amount_shares` worth of underlying tokens (WETH) to token holders. Note that this burns the shares held by the vault, but for accounting purposes, the `total_shares` variable isn't updated.

However, if a token holder chooses to liquidate his shares, his `shares_owned` are used entirely in both `alchemist.liquidate()` and `withdrawUnderlying()`. Because the contract no longer has fewer shares as a result of the yield distribution, the liquidation will fail.

POC

Refer to the `testVaultLiquidationAfterRepayment()` test case below. Note that this requires a fix to be applied for #2 first.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.18;

import "forge-std/Test.sol";
import "../../lib/utils/VyperDeployer.sol";

import "../IVault.sol";
import "../IAlchemistV2.sol";
import "../MintableERC721.sol";
import "openzeppelin/token/ERC20/IERC20.sol";

contract VaultTest is Test {
    ///@notice create a new instance of VyperDeployer
    VyperDeployer vyperDeployer = new VyperDeployer();

    FairFundingToken nft;
    IVault vault;
    address vaultAdd;
    IAlchemistV2 alchemist =
↪   IAlchemistV2(0x062Bf725dC4cDF947aa79Ca2aaCCD4F385b13b5c);
    IWhitelist whitelist =
↪   IWhitelist(0xA3dfCcbad1333DC69997Da28C961FF8B2879e653);
    address yieldToken = 0xa258C4606Ca8206D8aA700cE2143D7db854D168c;
    IERC20 weth = IERC20(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);
    // pranking from big WETH holder
    address admin = 0x2fEb1512183545f48f6b9C5b4EbfCaF49CfCa6F3;
    address user1 = address(0x123);
    address user2 = address(0x456);

    function setUp() public {
        vm.startPrank(admin);
        nft = new FairFundingToken();
        /// @notice: I modified vault to take in admin as a parameter
        /// because of pranking issues => setting permissions
        vault = IVault(
            vyperDeployer.deployContract("Vault", abi.encode(address(nft),
↪   admin))
        );
        // to avoid having to repeatedly cast to address
        vaultAdd = address(vault);
        vault.set_alchemist(address(alchemist));

        // whitelist vault and users in Alchemist system, otherwise will run
↪   into permission issues
        vm.stopPrank();
        vm.startPrank(0x9e2b6378ee8ad2A4A95Fe481d63CAba8FB0EBBF9);
        whitelist.add(vaultAdd);
        whitelist.add(admin);
        whitelist.add(user1);
        whitelist.add(user2);
        vm.stopPrank();

        vm.startPrank(admin);

        // add depositors
        vault.add_depositor(admin);
        vault.add_depositor(user1);
        vault.add_depositor(user2);

        // check yield token is whitelisted
        assert(alchemist.isSupportedYieldToken(yieldToken));

        // mint NFTs to various parties
        nft.mint(admin, 1);
        nft.mint(user1, 2);
        nft.mint(user2, 3);


        // give max WETH approval to vault & alchemist
        weth.approve(vaultAdd, type(uint256).max);
        weth.approve(address(alchemist), type(uint256).max);

        // send some WETH to user1 & user2
        weth.transfer(user1, 10e18);
        weth.transfer(user2, 10e18);

        // users give WETH approval to vault and alchemist
        vm.stopPrank();
        vm.startPrank(user1);
        weth.approve(vaultAdd, type(uint256).max);
        weth.approve(address(alchemist), type(uint256).max);
        vm.stopPrank();
        vm.startPrank(user2);
        weth.approve(vaultAdd, type(uint256).max);
        weth.approve(address(alchemist), type(uint256).max);
        vm.stopPrank();

        // by default, msg.sender will be admin
        vm.startPrank(admin);
    }

    function testVaultLiquidationAfterRepayment() public {
        uint256 depositAmt = 1e18;
        // admin does a deposit
        vault.register_deposit(1, depositAmt);
        vm.stopPrank();

        // user1 does a deposit too
        vm.prank(user1);
        vault.register_deposit(2, depositAmt);

        // simulate yield: someone does partial manual repayment
        vm.prank(user2);
        alchemist.repay(address(weth), 0.1e18, vaultAdd);

        // mark it as claimable (burn a little bit more shares because of rounding)
        vault.withdraw_underlying_to_claim(
            alchemist.convertUnderlyingTokensToShares(yieldToken, 0.01e18) + 100,
            0.01e18
        );
        vm.stopPrank();

        // user1 performs liquidation, it's fine
        vm.prank(user1);
        vault.liquidate(2, 0);

        // assert that admin has more shares than what the vault holds
        (uint256 shares, ) = alchemist.positions(vaultAdd, yieldToken);
        IVault.Position memory adminPosition = vault.positions(1);
        assertGt(adminPosition.sharesOwned, shares);

        vm.prank(admin);
        // now admin is unable to liquidate because of contract doesn't hold
↪   sufficient shares
        // expect Arithmetic over/underflow error
        vm.expectRevert(stdError.arithmeticError);
        vault.liquidate(1, 0);
    }
}
```

Impact

Failing liquidations as the contract attempts to burn more shares than it holds.

Code Snippet

https://github.com/sherlock-audit/2023-02-fair-funding/blob/main/fair-funding/contracts/Vault.vy#L341-L349 https://github.com/sherlock-audit/2023-02-fair-funding/blob/main/fair-funding/contracts/Vault.vy#L393-L404

Tool used

Foundry, Mainnet Forking, Manual Review

Recommendation

For the `shares_to_liquidate` and `amount_to_withdraw` variables, check against the vault's current shares and take the minimum of the 2.

The better fix would be to switch from marking yield claims with withdrawing WETH collateral to minting debt (alETH) tokens.

Discussion

Unstoppable-DeFi

This is correct.

We need to use `total_shares` and `position.shares_owned` to calculate the percentage of a positions contributions and then multiply it with the `remaining_shares` to receive the correct amount of shares during liquidation.

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/fair-funding/pull/9

Issue M-1: Migrator contract lacks sufficient permissions over vault positions

Source: https://github.com/sherlock-audit/2023-02-fair-funding-judging/issues/91

Found by

0x52, hickuphh3, Ruhum, minhtrng, jkoppel, ABA

Summary

The migrator contract lacks sufficient permissions over vault shares to successfully perform migration.

Vulnerability Detail

> Since vault potentially holds an Alchemix position over a long time during which changes at Alchemix could happen, the `migration_admin` has complete control over the vault and its position after giving depositors a 30 day window to liquidate (or transfer with a flashloan) their position if they're not comfortable with the migration.

We see that all that `migrate()` does is to trigger the `migrate()` function on the migration contract. However, no permissions over the vault's shares were given to the migration contract to enable it to say, liquidate to underlying or yield tokens. It also goes against what was intended, that is, *"complete control over the vault and its position"*.

Impact

Funds / positions cannot be successfully migrated due to lacking permissions.

Code Snippet

https://github.com/sherlock-audit/2023-02-fair-funding/blob/main/fair-funding/contracts/Vault.vy#L545-L556

Tool used

Manual Review

Recommendation

In addition to invoking the `migrate()` function, consider calling `approveWithdraw()` on the migrator contract for all of the vault's shares.

https://alchemix-finance.gitbook.io/v2/docs/alchemistv2#approvewithdraw

Also consider using `raw_call()` for this function call because the current `alchemist` possibly reverts, bricking the migration process entirely.

Discussion

Unstoppable-DeFi

This is correct, a `delegateCall` should have been used.

Will fix.

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/fair-funding/pull/6

Issue M-2: Broken Operator Mechanism: Just 1 malicious / compromised operator can permanently break functionality

Source: https://github.com/sherlock-audit/2023-02-fair-funding-judging/issues/46

Found by

weeeh_, hickuphh3, ABA, 0xSmartContract, rvierdiiev, csanuragjain

Summary

Operator access control isn't sufficiently resilient against a malicious or compromised actor.

Vulnerability Detail

I understand that we can assume all privileged roles to be trusted, but this is about the access control structure for the vault operators. The key thing here is that you can have multiple operators who can add or remove each other. As the saying goes, *"you are as strong as your weakest link"*, so all it required is for 1 malicious or compromised operator to permanently break protocol functionality, with no possible remediation as he's able to kick out all other honest operators, *including himself*

The vault operator can do the following:

1) Set the `alchemist` contract to any address (except null) of his choosing. He can therefore permanently brick the claiming and liquidation process, resulting in the permanent locking of token holders' funds in Alchemix.
2) Steal last auction funds. WETH approval is given to the `alchemist` contract every time `register_deposit` is called, and with the fact that anyone can settle the contract, the malicious operator is able to do the following atomically:
   - set the alchemist contract to a malicious implementation
     - contract returns a no-op + arbitrary `shares_issued` value when the `depositUnderlying()` function is called
   - settle the last auction (assuming it hasn't been)
   - pull auction funds from approval given
3) Do (1) and remove himself as an operator (ie. there are no longer any operators), permanently preventing any possible remediation.

Impact

DoS / holding the users' funds hostage.

Code Snippet

https://github.com/sherlock-audit/2023-02-fair-funding/blob/main/fair-funding/contracts/Vault.vy#L292-L300 https://github.com/sherlock-audit/2023-02-fair-funding/blob/main/fair-funding/contracts/Vault.vy#L589-L614

Tool used

Manual Review

Recommendation

Add an additional access control layer on top of operators: an `owner` that will be held by a multisig / DAO that's able to add / remove operators.

Discussion

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/fair-funding/pull/10

Issue M-3: Starting timestamp can be bypassed by calling `settle`

Source: https://github.com/sherlock-audit/2023-02-fair-funding-judging/issues/39

Found by

ck, Bahurum, carrot, 0xhacksmithh, 0xlmanini, HonorLt, 7siech, ABA, rvierdiiev, XKET, seyni

Summary

The starting timestamp set or still unset by the owner through the `start_auction` function can be bypassed by calling `settle`, which sends the first token to the fallback and then starts the auction for subsequent tokenIds.

Vulnerability Detail

The function `start_auction` is meant to be used to start the auction process, after which the bids start getting accepted. However, this entire system can be bypassed by calling the `settle` function. This leads to the first tokenId being minted to the fallback address, and the next tokenId auction being started immediately.

This can be exploited in two scenarios,

1. The function `start_auction` hasn't been called yet
2. The function `start_auction` has been called, and the timestamp passed is a timestamp in the future

In both these cases, the auctions can be made to start immediately. Thus the two issues are clubbed together.

The function `settle` only checks for the timestamp using the statement https://github.com/sherlock-audit/2023-02-fair-funding/blob/main/fair-funding/contracts/AuctionHouse.vy#L185 which is defined as https://github.com/sherlock-audit/2023-02-fair-funding/blob/main/fair-funding/contracts/AuctionHouse.vy#L247-L252 This check passes if the current timestamp is after the end of the epoch, but also if the current timestamp is before the start of the auction, which is the main issue here.

Inside the `settle` function, it sets the start and end timestamps properly, which allows bids to be made for subsequent tokenIds. https://github.com/sherlock-audit/2023-02-fair-funding/blob/main/fair-funding/contracts/AuctionHouse.vy#L200-L206

So even if the starting timestamp is unset or set in the future, the checks in `settle` pass, and the function then proceeds to write the start and end timestamps to process bids correctly.

Impact

Bids can be started immediately. This goes against the design of the protocol.

Code Snippet

The issue can be recreated with the following POC

```
def test_ATTACK_settle_before_start(owner, house, nft):
    token_id = house.current_epoch_token_id()
    assert house.highest_bidder() == pytest.ZERO_ADDRESS
    house.settle()
    house.bid(house.current_epoch_token_id(), house.RESERVE_PRICE())
    assert house.current_epoch_token_id() == token_id + 1
    assert nft.ownerOf(token_id) == owner
```

This shows a case where `start_action` is never called, yet the bids start. The same can be done if `start_auction` is called with a timestamp in the future

Tool used

Boa Manual Review

Recommendation

Change the check in `settle` to check for the end timestamp ONLY

Discussion

hrishibhat

While the issue is valid, there are no funds at risk with starting the auction early. Considering this issue a valid medium.

Unstoppable-DeFi

https://github.com/Unstoppable-DeFi/fair-funding/pull/7