SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:
Prepared by: Sherlock
Lead Security Expert: 0x52
Dates Audited: March 13 - March 27, 2023
Prepared on: April 11, 2023

Introduction

Y2K is a crypto-native take on structured products on-chain. The protocol creates liquid markets for hedging, leveraging, speculating and trading.

Scope

Earthquake @ 736b2e1e51bef6daa6a5ecd1decb7d156316d795

- Earthquake/src/v2/Carousel/Carousel.sol
- Earthquake/src/v2/Carousel/CarouselFactory.sol
- Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol
- Earthquake/src/v2/SemiFungibleVault.sol
- Earthquake/src/v2/TimeLock.sol
- Earthquake/src/v2/VaultFactoryV2.sol
- Earthquake/src/v2/VaultV2.sol
- Earthquake/src/v2/interfaces/ICarousel.sol
- Earthquake/src/v2/interfaces/ISemiFungibleVault.sol
- Earthquake/src/v2/interfaces/IVaultFactoryV2.sol
- Earthquake/src/v2/interfaces/IVaultV2.sol
- Earthquake/src/v2/interfaces/IWETH.sol
- Earthquake/src/v2/libraries/CarouselCreator.sol
- Earthquake/src/v2/libraries/VaultV2Creator.sol

Findings

Each issue has an assigned severity:

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

Issues found

Medium	High
23	5

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

rogueredwarf
kenzo
Dug
TrungOre
zeroknots
hklst4r
bin2chen
VAD37
berndartmueller
iglyx
Saeedalipoor01988
ABA
evan
Ch_301
cccz
0xvj
Delvir0
0x52
Ruhum
nobody2018
minhtrng
ltyu
0xnirlin
immeas
warRoom
mstpr-brainbot
p0wd3r
0xRobocop
libratus
ShadowForce
Respx
hickuphh3
toshii
ElKu
yixxas
spyrosonic10
holyhansss
sinarette
carrot
ast3ros
jprod15
Ace-30
HonorLt
Aymen0909
Bobface
Inspex
Bauer
charlesjhongc
GimelSec
Breeje
volodya
KingNFT
joestakey
J4de
datapunk
b4by_y0d4
twicek
AlexCzm
bulej93
Emmanuel
BPZ
climber2002
ni8mare
lemonmon
kaysoft
martin
peanuts
csanuragjain
ck
0Kage
0xmuxyz
shaka
auditor0517
jasonxiale
Junnon
ne0n
pfapostol
0xMojito
0xPkhatri

Issue H-1: Funds can be stolen because of incorrect update to `ownerToRollOverQueueIndex` for existing rollovers

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/2

Found by

kenzo, twicek, auditor0517, immeas, mstpr-brainbot, Ch_301, joestakey, minhtrng, 0xMojito, evan, TrungOre, libratus, spyrosonic10, yixxas, pfapostol, iglyx, Emmanuel, Junnon, Respx, csanuragjain, datapunk, toshii, HonorLt, climber2002, 0Kage, jasonxiale, ast3ros, Bauer, 0xnirlin, ne0n, bin2chen, VAD37, 0xRobocop, ck, charlesjhongc, rogueereddwarf, Aymen0909, volodya, ElKu, warRoom, AlexCzm, 0x52, holyhansss, sinarette, Dug, cccz, zeroknots, hickuphh3, berndartmueller, 0xPkhatri, ltyu, shaka

Summary

In the case where the owner has an existing rollover, the `ownerToRollOverQueueIndex` incorrectly updates to the last queue index. This causes the `notRollingOver` check to be performed on the incorrect `_id`, which then allows the depositor to withdraw funds that should've been locked.

Vulnerability Detail

In `enlistInRollover()`, if the user has an existing rollover, it overwrites the existing data:

```
if (ownerToRollOverQueueIndex[_receiver] != 0) {
  // if so, update the queue
  uint256 index = getRolloverIndex(_receiver);
  rolloverQueue[index].assets = _assets;
  rolloverQueue[index].epochId = _epochId;
```

However, regardless of whether the user has an existing rollover, the `ownerToRolloverQueueIndex` points to the last item in the queue:

```
ownerToRollOverQueueIndex[_receiver] = rolloverQueue.length;
```

Thus, the `notRollingOver` modifier will check the incorrect item for users with existing rollovers:

```
QueueItem memory item = rolloverQueue[getRolloverIndex(_receiver)];
if (
    item.epochId == _epochId &&
```

```
    (balanceOf(_receiver, _epochId) - item.assets) < _assets
) revert AlreadyRollingOver();
```

allowing the user to withdraw assets that should've been locked.

Impact

Users are able to withdraw assets that should've been locked for rollovers.

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L252-L257 https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L268 https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L755-L760

Tool used

Manual Review

Recommendation

The `ownerToRollOverQueueIndex` should be pointing to the last item in the queue in the `else` case only: when the user does not have an existing rollover queue item.

```
} else {
  // if not, add to queue
  rolloverQueue.push(
      QueueItem({
          assets: _assets,
          receiver: _receiver,
          epochId: _epochId
      })
  );
+ ownerToRollOverQueueIndex[_receiver] = rolloverQueue.length;
}
- ownerToRollOverQueueIndex[_receiver] = rolloverQueue.length;
```

Discussion

3xHarry

good catch

Issue H-2: Earlier users in rollover queue can grief later users

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/72

Found by

kenzo, twicek, mstpr-brainbot, immeas, minhtrng, TrungOre, evan, spyrosonic10, jprod15, Emmanuel, toshii, BPZ, Ruhum, ast3ros, bin2chen, rogueereddwarf, J4de, sinarette, nobody2018, Dug, hickuphh3, berndartmueller, Ch_301, ltyu

Summary

The current implementation enables users who are earlier in the queue to grief those who are later.

Vulnerability Detail

There is a `rolloverAccounting` mapping that, for every epoch, tracks the current index of the queue for which mints have been processed up to thus far.

When a user delists from the queue, the last user enlisted will replace the delisted user's queue index.

It is thus possible for the queue to be processed up to, or past, the delisted user's queue index, but before the last user has been processed, the processed user delists, thus causing the last user to not have his funds rollover.

POC

1. Alice enlists into the queue (index `1`), then Bob (index `2`)
2. Alice (or a relayer) calls `mintRollovers()` with `_operations = 1`, and Alice has her funds rollover.
3. Alice delists from the rollover.

Bob is then unable to have his funds rollover until the next epoch is created, unless he delists and re-enlists into the queue (defeating the purpose of rollover functionality).

Impact

Whether accidental or deliberate, it is possible for users to not have their funds rollover.

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L293-L296

Tool used

Manual Review

Recommendation

Instead of specifying the number of operations to execute, consider having start and end indexes, with a boolean mapping to track if a user's rollover has been processed.

Discussion

3xHarry

keeping track of rollovers with a mapping would increase gas cost substantially, however it would be a better solution than blocking delisting during deposit period

3xHarry

setting assets to 0 instead of removing the QueueItem from the array sounds like a more reasonable approach, given that it's very unlikely for the rollover queue array length to reach the max size. Also, there can be more markets with similar strike prices deployed at any time.

Issue H-3: `depositFee` can be bypassed via deposit queue

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/75

Found by

kenzo, immeas, minhtrng, TrungOre, evan, yixxas, iglyx, Respx, toshii, bulej93, Ruhum, ast3ros, VAD37, 0xRobocop, rogue_reddwarf, J4de, ElKu, AlexCzm, Inspex, Dug, hickuphh3, Ace-30

Summary

The deposit fee can be circumvented by a queue deposit + `mintDepositInQueue()` call in the same transaction.

Vulnerability Detail

A deposit fee is charged and increases linearly within the deposit window. However, this fee can be avoided if one deposits into the queue instead, then mints his deposit in the queue.

POC

Assume non-zero `depositFee`, valid epoch `_id = 1`. At epoch end, instead of calling `deposit(1, _assets, 0xAlice)`, Alice writes a contract that performs `deposit(0,_assets,0xAlice)` + `mintDepositInQueue(1,1)` to mint her deposit in the same tx (her deposit gets processed first because FILO system) . She pockets the `relayerFee`, essentially paying zero fees instead of incurring the `depositFee`.

Impact

Loss of protocol fee revenue.

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L494-L500 https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L332-L333 https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L354

Tool used

Manual Review

Recommendation

Because of the FILO system, charging the dynamic deposit fee will be unfair to queue deposits as they're reliant on relayers to mint their deposits for them. Consider taking a proportion of the relayer fee.

Discussion

3xHarry

This is a valid issue. We will apply depositFee to all mints (queue and direct). However, given that queue has the potential to affect when users's shares are minted because of FILO, min deposit has to be raised for the queue, to make it substantially harder to DDoS the queue. Minimizing DDoS queue deposits will lead to queue deposits getting the least fees as relayers can mint from the first second the epoch is created.

Issue H-4: When rolling over, user will lose his winnings from previous epoch

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/163

Found by

kenzo, mstpr-brainbot, immeas, minhtrng, TrungOre, evan, iglyx, toshii, carrot, bin2chen, VAD37, charlesjhongc, roguereddwarf, warRoom, Inspex, nobody2018, cccz, hickuphh3, berndartmueller, Ace-30

Summary

When `mintRollovers` is called, when the function mints shares for the new epoch for the user, the amount of shares minted will be the same as the original assets he requested to rollover - **not including the amount he won.** After this, **all these asset shares from the previous epoch are burnt.** So the user won't be able to claim his winnings.

Vulnerability Detail

When user requests to enlistInRollover, he supplies the amount of assets to rollover, and this is saved in the queue.

```
rolloverQueue[index].assets = _assets;
```

When `mintRollovers` is called, the function checks if the user won the previous epoch, and proceeds to **burn all the shares** the user requested to roll:

```
if (epochResolved[queue[index].epochId]) {
    uint256 entitledShares = previewWithdraw(
        queue[index].epochId,
        queue[index].assets
    );
    // mint only if user won epoch he is rolling over
    if (entitledShares > queue[index].assets) {
        ...
        // @note we know shares were locked up to this point
        _burn(
            queue[index].receiver,
            queue[index].epochId,
            queue[index].assets
        );
```

Then, and this is the problem, the function **mints to the user his original assets - `assetsToMint` - and not `entitledShares`.**

```
uint256 assetsToMint = queue[index].assets - relayerFee;
_mintShares(queue[index].receiver, _epochId, assetsToMint);
```

So the user has only rolled his original assets, but since all his share of them is burned, he will not be able anymore to claim his winnings from them.

Note that if the user had called `withdraw` instead of rolling over, all his shares would be burned, but he would receive his `entitledShares`, and not just his original assets. We can see in this in `withdraw`. Note that `_assets` is burned (like in minting rollover) but `entitledShares` is sent (unlike minting rollover, which only remints `_assets`.)

```
_burn(_owner, _id, _assets);
_burnEmissions(_owner, _id, _assets);
uint256 entitledShares;
uint256 entitledEmissions = previewEmissionsWithdraw(_id, _assets);
if (epochNull[_id] == false) {
    entitledShares = previewWithdraw(_id, _assets);
} else {
    entitledShares = _assets;
}
if (entitledShares > 0) {
    SemiFungibleVault.asset.safeTransfer(_receiver, entitledShares);
}
if (entitledEmissions > 0) {
    emissionsToken.safeTransfer(_receiver, entitledEmissions);
}
```

Impact

User will lose his rewards when rolling over.

Code Snippet

```
if (epochResolved[queue[index].epochId]) {
    uint256 entitledShares = previewWithdraw(
        queue[index].epochId,
        queue[index].assets
    );
    // mint only if user won epoch he is rolling over
    if (entitledShares > queue[index].assets) {
        ...
        // @note we know shares were locked up to this point
        _burn(
```

```
            queue[index].receiver,
            queue[index].epochId,
            queue[index].assets
        );
```

Tool used

Manual Review

Recommendation

Either remint the user his winnings also, or if you don't want to make him roll over the winnings, change the calculation so he can still withdraw his shares of the winnings.

Discussion

3xHarry

this makes total sense! thx for catching this!

3xHarry

will have to calculate how much his original deposit is worth in entitledShares and rollover the specified amount

Issue H-5: Adversary can break deposit queue and cause loss of funds

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/468

Found by

bin2chen, mstpr-brainbot, immeas, Respx, VAD37, joestakey, 0x52, 0xRobocop, HonorLt, nobody2018, rogueredwarf, ltyu, libratus, yixxas, jprod15, Bauer, Ruhum, warRoom

Summary

Vulnerability Detail

Carousel.sol#L531-L538

```
function _mintShares(
    address to,
    uint256 id,
    uint256 amount
) internal {
    _mint(to, id, amount, EMPTY);
    _mintEmissions(to, id, amount);
}
```

When processing deposits for the deposit queue, it _mintShares to the specified receiver which makes a _mint subcall.

ERC1155.sol#L263-L278

```
function _mint(address to, uint256 id, uint256 amount, bytes memory data)
↪   internal virtual {
    require(to != address(0), "ERC1155: mint to the zero address");

    address operator = _msgSender();
    uint256[] memory ids = _asSingletonArray(id);
    uint256[] memory amounts = _asSingletonArray(amount);

    _beforeTokenTransfer(operator, address(0), to, ids, amounts, data);

    _balances[id][to] += amount;
    emit TransferSingle(operator, address(0), to, id, amount);

    _afterTokenTransfer(operator, address(0), to, ids, amounts, data);
```

```
    _doSafeTransferAcceptanceCheck(operator, address(0), to, id, amount, data);
}
```

The base ERC1155 _mint is used which always behaves the same way that ERC721 safeMint does, that is, it always calls _doSafeTrasnferAcceptanceCheck which makes a call to the receiver. A malicious user can make the receiver always revert. This breaks the deposit queue completely. Since deposits can't be canceled this WILL result in loss of funds to all users whose deposits are blocked. To make matters worse it uses first in last out so the attacker can trap all deposits before them

Impact

Users who deposited before the adversary will lose their entire deposit

Code Snippet

Carousel.sol#L310-L355

Tool used

Manual Review

Recommendation

Override _mint to remove the safeMint behavior so that users can't DOS the deposit queue

Discussion

3xHarry

agree with this issue, there is no easy solution to this, as by definition when depositing into queue, the user gives up the atomicity of his intended mint. Looking at Openzeppelins 1155 implementation guide it is recommended to ensure the receiver of the asset is able to call safeTransferFrom. By removing the acceptance check in the `_mint` function, funds could be stuck in a smart contract.

Another alternative would be to do the 1155 acceptance check in the mint function and confiscate the funds if the receiver is not able to hold 1155s. The funds could be retrieved via a manual process from the treasury afterward.

3xHarry

going with Recommendation is prob the easiest way

Issue M-1: Deposit queue can be bricked if `relayerFee` is increased

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/1

Found by

auditor0517, immeas, mstpr-brainbot, minhtrng, p0wd3r, libratus, spyrosonic10, 0xvj, iglyx, Junnon, datapunk, jasonxiale, Ruhum, bin2chen, VAD37, rogueredwarf, ShadowForce, hickuphh3, Ch_301, Ace-30, shaka

Summary

Minting from the deposit queue could be bricked if there is an increase to the `relayerFee`.

Vulnerability Detail

The `relayerFee` can only be updated by the timelocker. Given the delay to update this parameter, an expected increase to the `relayerFee` can be frontrun by a queue deposit equal to the current `relayerFee`.

Attempting to mint the deposit reverts since the asset amount will be smaller than the relayer fee.

```
_mintShares(
  queue[i].receiver,
  _epochId,
  queue[i].assets - relayerFee
);
```

Because the deposit queue is FILO, all prior deposits cannot be processed.

Impact

Deposit queue may not be processed if an adversary is able to frontrun the execution of the relayer fee increase.

Another way of viewing the issue is that an adversary can prevent the `relayerFee` from ever increasing by depositing an amount equal to the current `relayerFee`.

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L334-L338

Tool used

Manual Review

Recommendation

Consider checking that the deposit queue lengths are zero in `changeRelayerFee()`.

P.S. `getDepositQueueLenght()` and `getRolloverQueueLenght()` have incorrect spelling.

Discussion

3xHarry

acknowledged

Issue M-2: `SemiFungibleVault` can be drained

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/45

Found by

Dug

Summary

The default implementation of `SemiFungibleVault`'s `withdraw` function allows anyone to drain all underlying tokens from the vault.

Vulnerability Detail

The `withdraw` function in `SemiFungibleVault` is defined as follows:

```
function withdraw(
    uint256 id,
    uint256 assets,
    address receiver,
    address owner
) external virtual returns (uint256 shares) {
    require(
        msg.sender == owner || isApprovedForAll(owner, msg.sender),
        "Only owner can withdraw, or owner has approved receiver for all"
    );

    shares = previewWithdraw(id, assets);

    _burn(owner, id, shares);

    emit Withdraw(msg.sender, receiver, owner, id, assets, shares);
    asset.safeTransfer(receiver, assets);
}
```

When a user calls `withdraw`, they provide an amount via the `assets` parameter. The `previewWithdraw` function is called, with the intention of using `amount` to calculate the number of shares that will be burned.

However, the default implementation of the `previewWithdraw` is a no-op that will always return `0`.

```
function previewWithdraw(
    uint256 id,
```

```
    uint256 assets
) public view virtual returns (uint256) {}
```

Therefore a user can call `withdraw` with any amount of `assets` and the `previewWithdraw` function will return `0`. This means that the `withdraw` function will burn `0` shares, and will transfer the full amount of `assets` to the `receiver` address.

Impact

A user can call `withdraw` with the vaults full balance as `assets` and drain all underlying tokens from the vault.

While I realize these functions are overridden in `VaultV2`, the fact that `SemiFungibleVault` is an abstract contract implies that it is meant to be reusable as a base contract for other vaults. Auditing it as an abstract contract, it is a vulnerable one.

The default implementation of this function is extremely dangerous should it be used in a different context.

Proof of concept

```
contract Derived is SemiFungibleVault {
    constructor(
        IERC20 asset_
    ) SemiFungibleVault(asset_, "name", "symbol", "tokenURI") {}
}

contract VaultIssues is Helper {
    function testDrain() public {
        // create a vault with 100 tokens
        address asset = address(new MintableToken("name", "symbol"));
        Derived vault = new Derived(IERC20(asset));
        deal(asset, address(vault), 100 ether, true);

        // confirm initial balances
        assertEq(IERC20(asset).balanceOf(address(vault)), 100 ether);
        assertEq(IERC20(asset).balanceOf(USER), 0);

        // drain the vault
        vm.prank(USER);
        vault.withdraw(0, 100 ether, USER, USER);

        // confirm drained vault
        assertEq(IERC20(asset).balanceOf(address(vault)), 0);
        assertEq(IERC20(asset).balanceOf(USER), 100 ether);
```

```
    }
}
```

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/SemiFungibleVault.sol#L66-L90

Tool used

Manual Review

Recommendation

The default implementation should be changed so the `assets` amount passed in gets burned.

```
-   shares = previewWithdraw(id, assets);

-   _burn(owner, id, shares);
+   _burn(owner, id, assets);
```

Alternatively, the `withdraw` and `previewWithdraw` functions should be marked as `abstract` and have no default implementation. This would force derived contracts to implement these functions and remove the default vulnerability.

Discussion

3xHarry

acknowledged, however, there is no plan to use any other vaults Earthquake VaultV2

Issue M-3: Admin can steal funds by setting 100 % withdraw fee

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/47

Found by

Bobface, VAD37, 0xRobocop, minhtrng, ShadowForce, Aymen0909, Ruhum, ElKu

Summary

The audit description mentions it is not accepted for the admin to be able to steal funds. However, the admin can set a 100 % withdraw fee for epochs, redirecting all user funds to the treasury.

Vulnerability Detail

In VaultFactoryV2.createEpoch(), the admin can supply a 100 % withdraw fee.

Impact

The withdraw fee causes all user deposits in a given epoch to be sent to the treasury upon the epoch end.

Tool used

Manual Review

Recommendation

Introduce a bounds check in VaultFactoryV2.createEpoch():

```
uint256 FEE_MAX = ... // some agreed upon value, e.g. 1 %
// ...
require(_withdrawalFee <= FEE_MAX, "fee too high");
```

Code Snippet

The PoC is written as Forge test. Paste the following code into `Earthquake/test/V2/HighFee.sol` and execute it with `forge test -m testHighFee`.

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

```
import "../../src/v2/VaultV2.sol";
import "../../src/v2/VaultFactoryV2.sol";
import "../../src/v2/Controllers/ControllerPeggedAssetV2.sol";

import "forge-std/Test.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract HighFee is Test {
    ERC20 token;
    VaultV2 premium;
    VaultV2 collateral;
    uint256 epochId;
    uint256 marketId;
    VaultFactoryV2 vaultFactory;
    ControllerPeggedAssetV2 controller;

    address treasury = address(2);
    address receiver = address(3);
    uint256 startDate = block.timestamp + 100;
    uint256 endDate = block.timestamp + 1_000;


    function testHighFee() external {
        // Create the token
        token = new ERC20("TOKEN", "TOKEN");

        // Create the factory
        vaultFactory = new VaultFactoryV2(address(1), address(this), treasury);

        // Create the controller
        controller = new ControllerPeggedAssetV2(address(vaultFactory),
↪   address(1), treasury);

        // Whitelist the controller
        vaultFactory.whitelistController(address(controller));

        // Create the market
        VaultFactoryV2.MarketConfigurationCalldata memory marketConfiguration =
            VaultFactoryV2.MarketConfigurationCalldata({
            token: address(token),
            strike: 1 ether,
            oracle: address(1),
            underlyingAsset: address(token),
            name: "",
            tokenURI: "",
            controller: address(controller)
        });
```

```
        (address _premium, address _collateral, uint256 _marketId) =
↪   vaultFactory.createNewMarket(marketConfiguration);
        premium = VaultV2(_premium);
        collateral = VaultV2(_collateral);
        marketId = _marketId;

        // Approve the token
        token.approve(address(premium), type(uint256).max);

        // Admin creates the epoch with 100 % withdraw fee
        (epochId, ) = vaultFactory.createEpoch(
            marketId,
            uint40(startDate),
            uint40(endDate),
            10_000 // <----
        );
    }
}
```

Discussion

3xHarry

agree

Issue M-4: ControllerPeggedAssetV2: outdated price may be used which can lead to wrong depeg events

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/70

Found by

kaysoft, bin2chen, Saeedalipoor01988, peanuts, 0xRobocop, carrot, minhtrng, rogueredwarf, evan, ShadowForce, TrungOre, p0wd3r, martin, ast3ros, lemonmon, Ch_301

Summary

The `updatedAt` timestamp in the price feed response is not checked. So outdated prices may be used.

Vulnerability Detail

The following checks are performed for the chainlink price feed: https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol#L299-L315

As you can see the `updatedAt` timestamp is not checked. So the price may be outdated.

Impact

The price that is used by the Controller can be outdated. This means that a depeg event may be caused due to an outdated price which is incorrect. Only current prices must be used to check for a depeg event.

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol#L273-L318

Tool used

Manual Review

Recommendation

Introduce a reasonable limit for how old the price can be and revert if the price is older:

```
iff --git a/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol
↪  b/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol
index 0587c86..cf2dcf5 100644
--- a/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol
+++ b/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol
@@ -275,8 +275,8 @@ contract ControllerPeggedAssetV2 {
             ,
             /*uint80 roundId*/
             int256 answer,
-            uint256 startedAt, /*uint256 updatedAt*/ /*uint80 answeredInRound*/
-            ,
+            uint256 startedAt,
+            uint256 updatedAt, /*uint80 answeredInRound*/

         ) = sequencerUptimeFeed.latestRoundData();

@@ -314,6 +314,8 @@ contract ControllerPeggedAssetV2 {

         if (answeredInRound < roundID) revert RoundIDOutdated();

+        if (updatedAt < block.timestamp - LIMIT) revert PriceOutdated();
+
         return price;
     }
```

Discussion

3xHarry

considering this

Issue M-5: Admin can steal funds from users by manipulating price oracle

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/73

Found by

zeroknots

Summary

The Y2K protocol is placing extensive trust on external price oracles. Y2k execute business logic such as depegs and vault balancing based on price signals from these oracles.

Y2k allows the admin to supply both the oracle address and the address of the insured ERC-20 contract. However, the contract does not validate these addresses, creating a vulnerability that can be exploited by a malicious admin to alter the outcome of the hedge bet and steal users' funds. This issue represents a severe security vulnerability that exposes users to financial risks, undermines the contract's trustworthiness, and potentially renders the contract unfit for its intended purpose.

A maliciously acting admin, could configure a new market with a manipulated oracle contract address and manipulate the outcome the vault's hedge position and thus rig the outcome of an epoch and drain users funds.

The protocol is designed to allow an Admin-EOA to configure new markets and epochs on those markets. The engagement scope defines following assumption:
Admin Should not be able to steal user funds

Vulnerability Detail

When creating new markets, no validation or checks against trusted price oracles is performed:

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/VaultFactoryV2.sol#L58-L74

A maliciously acting admin, could configure a new market with a **malicious** oracle contract that the admin controls. The admin can invest himself into a vault and bet against unexpecting Y2K users. At the end of the epoch, the admin manipulates the oracle contract in such a way, that it signals a severely broken peg.

The admin can then call triggerDepeg() on the controller, and collect a massive premium. https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol#L51-L62

ControllerPeggedAssetV2.sol Line 62 `int256 price = getLatestPrice(premiumVault.token());` is in complete control of the admin.

Impact

Loss of user funds Reputation Damage

Code Snippet

Exploit Contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract OracleExploit is AggregatorV3Interface {
    AggregatorV3Interface public realOracle;


    struct OraclePriceFeed {
        bool enabled;
        uint80 roundId;
        int256 answer;
        uint256 startedAt;
        uint256 updatedAt;
        uint80 answeredInRound;
    }

    OraclePriceFeed pwned;

    constructor(address _oracle) {
        realOracle = AggregatorV3Interface(_oracle);
    }

    function setPwned(OraclePriceFeed calldata _data) public {
        pwned = _data;
    }

    function decimals() external view override returns (uint8) {
        return realOracle.decimals();
    }

    function description() external view override returns (string memory) {
        return realOracle.description();
    }

    function version() external view override returns (uint256) {
        return realOracle.version();
    }
```

```
    function getRoundData(uint80 _roundId)
        external
        view
        override
        returns (uint80 roundId, int256 answer, uint256 startedAt, uint256
↪   updatedAt, uint80 answeredInRound)
    {
        if (pwned.enabled) {
            return (pwned.roundId, pwned.answer, pwned.startedAt,
↪   pwned.updatedAt, pwned.answeredInRound);
        } else {
            return realOracle.getRoundData(_roundId);
        }
    }

    function latestRoundData()
        external
        view
        override
        returns (uint80 roundId, int256 answer, uint256 startedAt, uint256
↪   updatedAt, uint80 answeredInRound)
    {
        if (pwned.enabled) {
            return (pwned.roundId, pwned.answer, pwned.startedAt,
↪   pwned.updatedAt, pwned.answeredInRound);
        } else {
            return realOracle.latestRoundData();
        }
    }
}
```

Tool used

Manual Review

Recommendation

Since Y2K is placing so much trust on oracles, adequate validation processes such as whitelisting via the TimeLock should be implemented

Discussion

3xHarry

considering this

dmitriia

Leaving 73 and 74 as one separate medium to represent issues sourced from the root cause of Admin supplying core Vault parameters.

Issue M-6: Amount of emissionTokens are stucked in the vault due to rounding calculation

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/89

Found by

TrungOre

Summary

Using round-down to calculate amount of distributing emissionTokens make some emissionTokens stuck in the contract

Vulnerability Detail

Implementation. of function `Carousel.previewEmissionWithdraw()` is as follows:

It was used in function `Carousel.withdraw()` to calculate the emissions to withdraw. Assume that `_assets * emissions[_id]` isn't divisible by `finalTVL`, an amount of emissionTokens won't be transfered to users. For instance,

- `_asset = 9, emissions[_id] = 15, finalTVL = 10 --> entitleAmount = 9 * 15 / 10 = 13.5 > 13`

The amount of emissionTokens is freezed in vault contract can reach to nearly`finalTVL` for each epoch. Note that this issue also happens for function `VaultV2.previewWithdraw()`.

Impact

Amount of emissionTokens are freezed in the vault contracts.

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L630-L636

Tool used

Manual review

Recommendation

Create a new function to let the owner withdraw the remaining amount of emissionToken out of vault

Discussion

3xHarry

considering this, however, mulDivDown was chosen on purpose, to prevent revert by underflow. It's more important that the withdraw function is not reverting instead of leaving some dust, especially bc its used in the while loops.

Issue M-7: Carousel: `getRolloverTVL` function returns ERC1155 balance instead of TVL

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/95

Found by

roguereddwarf

Summary

The `Carousel.getRolloverTVL` function is supposed to return the total value locked in the rollover queue for a given `_epochId`.

The issue is that this function returns the wrong result. It is supposed to return the total value locked, i.e. the underlying assets of the Vault. However it returns the ERC1155 vault tokens.

Vulnerability Detail

The function just sums up the `assets` field of each queue item: https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L668

However 1 vault token might be worth e.g. 2 WETH.

The number of underlying tokens that `assets` is worth needs to be calculated as:

```
previewWithdraw(
    rolloverQueue[i].epochId,
    rolloverQueue[i].assets
)
```

Impact

The returned TVL is wrong which leads to wrong calculations in any components relying on this function.

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L655-L671

Tool used

Manual Review

Recommendation

Fix:

```
diff --git a/Earthquake/src/v2/Carousel/Carousel.sol
 ↪  b/Earthquake/src/v2/Carousel/Carousel.sol
index ed460af..44fcdae 100644
--- a/Earthquake/src/v2/Carousel/Carousel.sol
+++ b/Earthquake/src/v2/Carousel/Carousel.sol
@@ -665,7 +665,7 @@ contract Carousel is VaultV2 {
                     rolloverQueue[i].assets
                 ) > rolloverQueue[i].assets)
             ) {
-                tvl += rolloverQueue[i].assets;
+                tvl +=
 ↪  previewWithdraw(rolloverQueue[i].epochId,rolloverQueue[i].assets);
             }
         }
     }
```

Discussion

3xHarry

great catch

Issue M-8: ControllerPeggedAssetV2: `triggerEndEpoch` function can be called even if epoch is null epoch leading to loss of funds

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/108

Found by

0xnirlin, bin2chen, kenzo, holyhansss, climber2002, minhtrng, rogue reddwarf, evan, charlesjhongc, ltyu, libratus, 0xvj, berndartmueller, KingNFT, warRoom

Summary

An epoch can be resolved in three ways which correspond to the three functions available in the Controller: `triggerDepeg`, `triggerEndEpoch`, `triggerNullEpoch`.

The issue is that `triggerEndEpoch` can be called even though `triggerNullEpoch` should be called. "Null epoch" means that any of the two vaults does not have funds deposited. In this case the epoch should be resolved with `triggerNullEpoch` such that funds are not transferred from the premium vault to the collateral vault.

So in `triggerEndEpoch` is should be checked whether the conditions for a null epoch apply. If that's the case, the `triggerEndEpoch` function should revert.

Vulnerability Detail

The assumption the code makes is that if the null epoch applies, `triggerNullEpoch` will be called before the end timestamp of the epoch which is when `triggerEndEpoch` can be called.

This is not necessarily true.

`triggerNullEpoch` might not be called in time (e.g. because the epoch duration is very short or simply nobody calls it) and then the `triggerEndEpoch` function can be called which sends the funds from the premium vault into the collateral vault: https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol#L172-L192

If the premium vault is the vault which has funds and the collateral vault does not, then the funds sent to the collateral vault are lost.

Impact

Loss of funds for users that have deposited into the premium vault.

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol#L144-L202

Tool used

Manual Review

Recommendation

`triggerEndEpoch` should only be callable when the conditions for a null epoch don't apply:

```
diff --git a/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol
↪  b/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol
index 0587c86..7b25cf3 100644
--- a/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol
+++ b/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol
@@ -155,6 +155,13 @@ contract ControllerPeggedAssetV2 {
             collateralVault.epochExists(_epochId) == false
         ) revert EpochNotExist();

+        if (
+            premiumVault.totalAssets(_epochId) == 0 ||
+            collateralVault.totalAssets(_epochId) == 0
+        ) {
+            revert VaultZeroTVL();
+        }
+
         (, uint40 epochEnd, ) = premiumVault.getEpochConfig(_epochId);

         if (block.timestamp <= uint256(epochEnd)) revert EpochNotExpired();
```

Issue M-9: Controller doesn't send treasury funds to the vault's treasury address

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/110

Found by

bin2chen, 0xnirlin, nobody2018, Dug, rogue reddwarf, Ruhum

Summary

The Controller contract sends treasury funds to its own immutable `treasury` address instead of sending the funds to the one stored in the respective vault contract.

Vulnerability Detail

Each vault has a treasury address that is assigned on deployment which can also be updated through the factory contract:

But, the Controller, responsible for sending the fees to the treasury, uses the immutable treasury address that it was initialized with:

Impact

It's not possible to have different treasury addresses for different vaults. It's also not possible to update the treasury address of a vault although it has a function to do that. Funds will always be sent to the address the Controller was initialized with.

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/VaultV2.sol#L79 https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/VaultV2.sol#L265-L268

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol#L186
https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol#L40

Tool used

Manual Review

Recommendation

The Controller should query the Vault to get the correct treasury address, e.g.:

Discussion

3xHarry

will use one location for the treasury address which will be on the factory.

Issue M-10: Stuck emissions for nullified epochs

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/122

Found by

kenzo, bin2chen, immeas, 0x52, sinarette, carrot, roguereddwarf, cccz, libratus, hickuphh3, Ch_301, ltyu

Summary

If either the premium and / or collateral vault has 0 TVL for an epoch with emissions, those emissions will not be withdrawable by anyone.

Vulnerability Detail

The `finalTVL` set for a vault with 0 TVL (epoch will be nullified) will be 0. As a result, emissions that were allocated to that vault are not withdrawable by anyone.

It's admittedly unlikely to happen since the `emissionsToken` is expected to be Y2K which has value and is tradeable.

Impact

Emissions cannot be recovered.

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L157 https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L630-L636

Tool used

Manual Review

Recommendation

Create a function to send emissions back to the treasury if an epoch is marked as nullified.

A related issue is that if both the premium and collateral vaults have 0 TVL, only the collateral vault gets marked as nullified. Consider handling this edge case.

Discussion

3xHarry

great catch

Issue M-11: PremiumVault only depositor can refuse to give money to the couterpart

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/135

Found by

hklst4r

Summary

If someone is the only depositor of the Premium Vault of an epoch, he can refuse to give his/her money to the counterpart.

Vulnerability Detail

Let's suppose in some epoch, there is only one address depositing into the Premium vault and there is no depeg event during the epoch. At the end of the epoch, normally all funds in the Premium Vault will be sent to the treasury and the collateral Vault. However, the only address who deposited into the Premium vault can do as following to avoid paying:(`_id` is the id of this epoch)

1. transfer all the ERC1155(id = `_id`) (s)he owns to `address(0)`
2. Such action will trigger the `_beforeTokenTransfer` function in contract ERC1155Supply, which then sets totalsupply to `0`.
3. Then (s)he can call `triggerNullEpoch` function of the Controller, as the `totalAssets` now is actually `0`.
4. The treasury amd the collateral Vault depositors won't get paid.

Impact

If someone is the only depositor of the Premium Vault of an epoch, he can refuse to give his/her money to the counterpart. I am submitting this issue as meduim because the condition is not easily satisfied and the attacker can not get profit through it (but harm others' profits):

1. The Premium Vault should have only one person depositing in that epoch.(or all depositors collude)
2. Whether the attacker perform attacks like this or not, he loses everthing he deposits, but he can harm others by doing this.

Code Snippet

1. `_beforeTokenTransfer` in file: ERC1155Supply.sol line 36, When transfer to zero-address is called, `_totalSupply[id]` will reduce.(When all deposited money are transfered to 0 address, `_totalSupply[id]` goes to zero)

```
    /**
 * @dev See {ERC1155-_beforeTokenTransfer}.
 */
function _beforeTokenTransfer(
    address operator,
    address from,
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
) internal virtual override {
    super._beforeTokenTransfer(operator, from, to, ids, amounts, data);

    if (from == address(0)) {
        for (uint256 i = 0; i < ids.length; ++i) {
            _totalSupply[ids[i]] += amounts[i];
        }
    }

    if (to == address(0)) {
        for (uint256 i = 0; i < ids.length; ++i) {
            uint256 id = ids[i];
            uint256 amount = amounts[i];
            uint256 supply = _totalSupply[id];
            require(supply >= amount, "ERC1155: burn amount exceeds
↪  totalSupply");
            unchecked {
                _totalSupply[id] = supply - amount;
            }
        }
    }
}
```

2. `triggerNullEpoch` function in the file: `ControllerPeggedAssetV2.sol`, line 232-243 & `totalAssets` function in the file `vaultV2.sol` line 202. After transfering all funds to zero address, the attacker can call the `triggerNullEpoch` function. https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol#L232-L243

Tool used

Manual Review (vscode)

Recommendation

I suggest adding check for burning(transfering to zero-address). After the epoch ends, no one can burn any ERC1155 tokens. This can be done overriding function `_beforeTokenTransfer` in VaultV2 and add end-epoch check when burning tokens.

Discussion

3xHarry

the recommended check however would also make burning tokens possible at any other state of the epoch. In general burning, assets need to be possible after epoch is settled, to be able to roll over user deposits. The correct implementation would be to revert on transfer as burning should only be done by smart contract.

3xHarry

Having a second look at this issue, i saw that the openzeppelin implementation prohibits burning your tokens (sending to address(0)) using the safeTransferFrom() method. Burning tokens can only be done in withdraw function which can only be called once epoch is resolved. Therefore this issue is invalid.

open

Issue M-12: Inadequate price oracle checks

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/154

Found by

ABA, Saeedalipoor01988

Summary

Price of premium vault token, when triggering a depeg, is taken via Chainlink's `latestRoundData` function. Not all checks are not on the `latestRoundData` output, thus leaving a possibility for the price to be outdated or have suffered a price manipulation that in turn would go undetected.

Concrete the issues are:

1. Missing outdated data validation on `latestRoundData`

There is not checked if the answer was received from `latestRoundData` was given an accepted time window.

Note: This is different from the sequencer's uptime, where there is a check in place.

2. No resistance for oracle price manipulation

This missing check consists of saving previously received price and compare it with the new price. If the difference is above a certain threshold then stop the execution.

Vulnerability Detail

For the second issue, see Summary, for the first issue, in `ControllerPeggedAssetV2` the price for premium vault tokens when triggering a depeg is retrieved via the `getLatestPrice` function.

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol#L273

```
function getLatestPrice(address _token) public view returns (int256) {
```

`getLatestPrice` retrieves the Chainlink feed price via `latestRoundData` and does several checks. What it does not check is if the retrieved price is a stale one.

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Controllers/ControllerPeggedAssetV2.sol#L299-L318

```
    (uint80 roundID, int256 price, , , uint80 answeredInRound) = priceFeed
        .latestRoundData();
    uint256 decimals = priceFeed.decimals();
```

```
    if (decimals < 18) {
        decimals = 10**(18 - (decimals));
        price = price * int256(decimals);
    } else if (decimals == 18) {
        price = price;
    } else {
        decimals = 10**((decimals - 18));
        price = price / int256(decimals);
    }


    if (price <= 0) revert OraclePriceZero();


    if (answeredInRound < roundID) revert RoundIDOutdated();


    return price;
}
```

`latestRoundData`'s 4th return value is `updatedAt`: *Timestamp of when the round was updated.* https://docs.chain.link/data-feeds/api-reference/#latestrounddata

This value is not stored or checked for an outdated price.

Another, not so common check relating to time is to see if the round was incomplete, by checking if `updateTime` is 0.

Impact

The price impacts where or not a trigger depeg call reaches the strike price or not, this ultimately means the correct execution of the protocol functionality.

Code Snippet

Tool used

Manual Review

Recommendation

For issue 1:

- when launching the `ControllerPeggedAssetV2` contract, also include a `priceUpdateThreshold` variable that stores what is the tolerated age (in seconds) of the retrieved price.

- save the `updatedAt` return data from `latestRoundData`
- check it to be != 0
- also check that it was determined less then `priceUpdateThreshold` seconds ago

For issue 2:

- for each token oracle save a `previousValidPrice` while also providing a deviation threshold for which to accept a new price.
- the threshold can be set as to not impact a potential black swan event that would cause a sudden dip in prices.

Discussion

3xHarry

We will consider this, however, given that the controller should be able to read multiple chainlink oracles, one oracle could have a different heartbeat which makes defining a priceUpdateThreshold hard.

dmitriia

To clarify, this is about price stability in general and cannot be a dup of mere `updatedAt > 0` issues, which are best practice suggestions.

Issue M-13: User deposit may never be entertained from deposit queue

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/174

Found by

twicek, immeas, minhtrng, TrungOre, evan, jprod15, csanuragjain, Respx, 0Kage, Ruhum, Bauer, bin2chen, 0xmuxyz, ck, rogueredwarf, ElKu, nobody2018, ShadowForce, hickuphh3, Ace-30

Summary

Due to FILO (first in last out) stack structure, while dequeuing, the first few entries may never be retrieved. These means User deposit may never be entertained from deposit queue if there are too many deposits

Vulnerability Detail

1. Assume User A made a deposit which becomes 1st entry in `depositQueue`
2. Post this X more deposits were made, so `depositQueue.length=X+1`
3. Relayer calls `mintDepositInQueue` and process `X-9` deposits

```
while ((length - _operations) <= i) {
            // this loop impelements FILO (first in last out) stack to reduce gas
 ↪  cost and improve code readability
            // changing it to FIFO (first in first out) would require more code
 ↪  changes and would be more expensive
            _mintShares(
                queue[i].receiver,
                _epochId,
                queue[i].assets - relayerFee
            );
            emit Deposit(
                msg.sender,
                queue[i].receiver,
                _epochId,
                queue[i].assets - relayerFee
            );
            depositQueue.pop();
            if (i == 0) break;
            unchecked {
                i--;
```

```
            }
        }
```

4. This reduces deposit queue to only 10
5. Before relayer could process these, Y more deposits were made which increases deposit queue to `y+10`
6. This means Relayer might not be able to again process User A deposit as this deposit is lying after processing `Y+9` deposits

Impact

User deposit may remain stuck in deposit queue if a large number of deposit are present in queue and relayer is interested in dequeuing all entries

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L310

Tool used

Manual Review

Recommendation

Allow User to dequeue deposit queue based on index, so that if such condition arises, user would be able to dequeue his deposit (independent of relayer)

Discussion

3xHarry

depositing into queue should count as committing to an epoch. By giving the user the ability to delist his queue he could take advantage of market movements. However, we will raise min deposit for the queue to make DDoS very expensive.

Issue M-14: Ineffective timelocker due to epoch length

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/181

Found by

kenzo

Summary

The timelocker has a timelock of 3 days, while the epochs length is 7 or 30 days. Users can not withdraw their funds after an epoch has started. This renders the timelock ineffective, as users can not withdraw their funds while a malicious timelock transaction is pending, and the admin can rug the protocol.

Vulnerability Detail

The Y2K docs state that epochs locking period will be weekly and monthly. The contest readme states that `Admin Should not be able to steal user funds`. To mitigate against such risk, a timelocker has been added. It sets the waiting period as such:

```
uint32 public constant MIN_DELAY = 3 days;
uint32 public constant MAX_DELAY = 30 days;
uint32 public constant GRACE_PERIOD = 14 days;
function queue(
    ...
    if (
        _timestamp < block.timestamp + MIN_DELAY ||
        _timestamp > block.timestamp + MAX_DELAY
    ) {
        revert TimestampNotInRangeError(block.timestamp, _timestamp);
    }
```

So a transaction has to wait for a minimum of 3 days. But since epochs last for a week or a month (as per the documentation), a user that has locked his funds for such a period can not withdraw them in the 3 days of timelock. He can not effectively do anything to save his funds in case of malicious or compromised admin.

The contest readme states that `Admin Should not be able to steal user funds`. But for example an admin can rug in the following way: Once an epoch has started, queue a tx that whitelists his own address for the vault (`whitelistAddressOnMarket`), then queue a tx that whitelists his EOA as controller (`whitelistController`), then queue a tx that changes the controller to his EOA (`changeController`), and after 3

days, he'll execute them all, and use his EOA to call `sendTokens` in the vault and send himself all the user funds.

Impact

Timelock is not effective. Admin can rug user funds, which according to the contest readme, should not be possible.

Code Snippet

```
uint32 public constant MIN_DELAY = 3 days;
uint32 public constant MAX_DELAY = 30 days;
uint32 public constant GRACE_PERIOD = 14 days;
function queue(
    ...
    if (
        _timestamp < block.timestamp + MIN_DELAY ||
        _timestamp > block.timestamp + MAX_DELAY
    ) {
        revert TimestampNotInRangeError(block.timestamp, _timestamp);
    }
```

Tool used

Manual Review

Recommendation

I believe that for the timelock to be effective, it has to have a minimum delay which is longer than the epoch's duration.

Discussion

3xHarry

considering this

Issue M-15: changeTreasury() Lack of check and remove old

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/208

Found by

bin2chen, VAD37

Summary

changeTreasury() Lack of check and remove old

Vulnerability Detail

changeTreasury() used to set new treasury The code is as follows

```
function changeTreasury(uint256 _marketId, address _treasury)
    public
    onlyTimeLocker
{
    if (_treasury == address(0)) revert AddressZero();

    address[2] memory vaults = marketIdToVaults[_marketId];

    if (vaults[0] == address(0) || vaults[1] == address(0)) {
        revert MarketDoesNotExist(_marketId);
    }
    IVaultV2(vaults[0]).whiteListAddress(_treasury);
    IVaultV2(vaults[1]).whiteListAddress(_treasury);
    IVaultV2(vaults[0]).setTreasury(treasury);
    IVaultV2(vaults[1]).setTreasury(treasury);

    emit AddressWhitelisted(_treasury, _marketId);
}
```

The above code has the following problem:

1. no check whether the new treasury same as the old. If it is the same, the whitelist will be canceled.
2. Use setTreasury(VaultFactoryV2.treasury), it should be setTreasury(_treasury)
3. not cancel old treasury from the whitelist

Impact

whiteListAddress abnormal

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/VaultFactoryV2.sol#L228

Tool used

Manual Review

Recommendation

```
    function changeTreasury(uint256 _marketId, address _treasury)
        public
        onlyTimeLocker
    {
        if (_treasury == address(0)) revert AddressZero();

        address[2] memory vaults = marketIdToVaults[_marketId];

        if (vaults[0] == address(0) || vaults[1] == address(0)) {
            revert MarketDoesNotExist(_marketId);
        }

+       require(vaults[0].treasury() !=_treasury,"same"); //check same
+       IVaultV2(vaults[0]).whiteListAddress(vaults[0].treasury()); //cancel old
 ↪  whitelist
+       IVaultV2(vaults[1]).whiteListAddress(vaults[1].treasury()); //cancel old
 ↪  whitelist

        IVaultV2(vaults[0]).whiteListAddress(_treasury);
        IVaultV2(vaults[1]).whiteListAddress(_treasury);
+       IVaultV2(vaults[0]).setTreasury(_treasury);
+       IVaultV2(vaults[1]).setTreasury(_treasury);
-       IVaultV2(vaults[0]).setTreasury(treasury);
-       IVaultV2(vaults[1]).setTreasury(treasury);

        emit AddressWhitelisted(_treasury, _marketId);
    }
```

Discussion

dmitriia

Keeping it separate from 435 because of whitelist observation (1)

Issue M-16: mintRollovers should require entitledShares >= relayerFee

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/293

Found by

cccz, roguereddwarf, iglyx

Summary

mintRollovers should require entitledShares >= relayerFee

Vulnerability Detail

In mintRollovers, the rollover is only not skipped if queue[index].assets >= relayerFee,

```
if (entitledShares > queue[index].assets) {
    // skip the rollover for the user if the assets cannot cover the relayer fee
↪   instead of revert.
    if (queue[index].assets < relayerFee) {
        index++;
        continue;
    }
```

In fact, since the user is already profitable, entitledShares is the number of assets of the user, which is greater than queue[index].assets, so it should check that entitledShares >= relayerFee, and use entitledShares instead of queue[index].assets to subtract relayerFee when calculating assetsToMint later.

Impact

This will prevent rollover even if the user has more assets than relayerFee

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/Carousel/Carousel.sol#L401-L406

Tool used

Manual Review

Recommendation

Change to

```
                if (entitledShares > queue[index].assets) {
                    // skip the rollover for the user if the assets cannot cover
 ↪  the relayer fee instead of revert.
-                   if (queue[index].assets < relayerFee) {
+                   if (entitledShares < relayerFee) {
                        index++;
                        continue;
                    }
...
-                   uint256 assetsToMint = queue[index].assets - relayerFee;
+                   uint256 assetsToMint = entitledShares - relayerFee;
```

Issue M-17: The owner won't be able to create two Market with the same `token` ,`strike` and different ERC20 for deposit

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/300

Found by

Delvir0, Ch_301, 0xvj

Summary

One of the changes compared to V1. deposit asset can be any erc20

Vulnerability Detail

On createNewCarouselMarket() The owner can't create two markets for the same `token` and `strike` with different `underlyingAsset`

e.g. in case there is a Market with `token == x` ,`striek == y` and `underlyingAsset == WETH`. The owner won't be able to create another Market with `token == x` ,`striek == y` and `underlyingAsset == e.g.USDC`

Impact

The owner won't be able to create two Market with the same `token` ,`strike` and different ERC20 for deposit

Code Snippet

Tool used

Manual Review

Recommendation

```
    function createNewCarouselMarket(
        CarouselMarketConfigurationCalldata memory _marketCalldata
    )
        external
        onlyOwner
        returns (
            address premium,
            address collateral,
            uint256 marketId
```

```
        )
    {
        if (!controllers[_marketCalldata.controller]) revert ControllerNotSet();
        if (_marketCalldata.token == address(0)) revert AddressZero();
        if (_marketCalldata.oracle == address(0)) revert AddressZero();
        if (_marketCalldata.underlyingAsset == address(0)) revert AddressZero();

        if (tokenToOracle[_marketCalldata.token] == address(0)) {
            tokenToOracle[_marketCalldata.token] = _marketCalldata.oracle;
        }
-     marketId = getMarketId(_marketCalldata.token, _marketCalldata.strike);
+     marketId = getMarketId(_marketCalldata.token, _marketCalldata.strike,
 ↪   _marketCalldata.underlyingAsset);
```

Discussion

3xHarry

this makes sense

Issue M-18: `TimeLock.execute` lacks payable

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/387

Found by

kenzo, bin2chen, mstpr-brainbot, 0x52, GimelSec, rogueereddwarf, ltyu, Breeje, warRoom

Summary

`TimeLock.execute` lacks `payable`. If `_value` in `TimeLock.execute` is not zero, it could always revert.

Vulnerability Detail

`TimeLock.execute` lacks `payable`. The caller cannot send the value. https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/TimeLock.sol#L113

```
function execute(
    address _target,
    uint256 _value,
    string calldata _func,
    bytes calldata _data,
    uint256 _timestamp
) external onlyOwner returns (bytes memory) {


    // call target
    (bool ok, bytes memory res) = _target.call{value: _value}(data);

}
```

And the contract is modified from https://solidity-by-example.org/app/time-lock/ . The example code has the `payable` receive function. But `TimeLock` doesn't have one.

Impact

`TimeLock.execute` cannot work if `_value` != 0.

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/main/Earthquake/src/v2/TimeLock.sol#L113

Tool used

Manual Review

Recommendation

Add `payable` on `TimeLock.execute`. Also add a check to ensure `msg.value == _value`.

```
function execute(
    address _target,
    uint256 _value,
    string calldata _func,
    bytes calldata _data,
    uint256 _timestamp
) external payable onlyOwner returns (bytes memory) {


    // call target
    (bool ok, bytes memory res) = _target.call{value: _value}(data);

}
```

Discussion

3xHarry

valid issue

Issue M-19: `Carousel.mintRollovers` potentially mints `0` shares and can grief rollover queue

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/418

Found by

berndartmueller, kenzo, evan

Summary

If the deposited assets for a queued rollover item are equal to the relayer fee, the rollover will be minted with `0` shares, potentially leading to zero TVL and hence `finalTVL[_id] = 0`. This will cause the `previewWithdraw` call to revert due to division by zero and the rollover queue will be stuck forever.

Vulnerability Detail

Minting rollovers in the carousel vault iterates over all items in the `rolloverQueue` queue. Each item is processed, and the entitled shares (`entitledShares`) are calculated using `previewWithdraw`. If the
entitled shares are greater than the deposited assets), the rollover is minted.

However, if the deposited assets for the queued item are equal to the relayer fee, the assets to mint (`assetsToMint`) calculated in line 436 will be `0`.

If this happens to be the only deposit (mint) for the epoch and the vaults TVL remains zero, the `previewWithdraw` call in line 396 will revert due to division by zero.

Impact

Once there is a rollover minted with `0` shares for an epoch and the vaults TVL (i.e., `finalTVL`) remains zero, the rollover queue will be stuck forever unless the owner of this queue item delists it.

Code Snippet

src/v2/Carousel/Carousel.mintRollovers

```
361: function mintRollovers(uint256 _epochId, uint256 _operations)
362:     external
363:     epochIdExists(_epochId)
364:     epochHasNotStarted(_epochId)
365:     nonReentrant
366: {
...      // [...]
```

```
392:
393:     while ((index - prevIndex) < (_operations)) {
394:         // only roll over if last epoch is resolved
395:         if (epochResolved[queue[index].epochId]) {
396: @>          uint256 entitledShares = previewWithdraw( // @audit-info reverts if epoch's `finalTVL` == 0
397:                 queue[index].epochId,
398:                 queue[index].assets
399:             );
400:             // mint only if user won epoch he is rolling over
401:             if (entitledShares > queue[index].assets) {
402:                 // skip the rollover for the user if the assets cannot cover the relayer fee instead of revert.
403:                 if (queue[index].assets < relayerFee) {
404:                     index++;
405:                     continue;
406:                 }
407:                 // @note we know shares were locked up to this point
408:                 _burn(
409:                     queue[index].receiver,
410:                     queue[index].epochId,
411:                     queue[index].assets
412:                 );
413:                 // transfer emission tokens out of contract otherwise user could not access them as vault shares are burned
414:                 _burnEmissions(
415:                     queue[index].receiver,
416:                     queue[index].epochId,
417:                     queue[index].assets
418:                 );
419:                 // @note emission token is a known token which has no before transfer hooks which makes transfer safer
420:                 emissionsToken.safeTransfer(
421:                     queue[index].receiver,
422:                     previewEmissionsWithdraw(
423:                         queue[index].epochId,
424:                         queue[index].assets
425:                     )
426:                 );
427:
428:                 emit Withdraw(
429:                     msg.sender,
430:                     queue[index].receiver,
431:                     queue[index].receiver,
432:                     _epochId,
433:                     queue[index].assets,
434:                     entitledShares
435:                 );
```

```
436: @>                 uint256 assetsToMint = queue[index].assets - relayerFee; //
↪  @audit-info `assetsToMint` can potentially become 0
437:                    _mintShares(queue[index].receiver, _epochId, assetsToMint);
438:                    emit Deposit(
439:                        msg.sender,
440:                        queue[index].receiver,
441:                        _epochId,
442:                        assetsToMint
443:                    );
444:                    rolloverQueue[index].assets = assetsToMint;
445:                    rolloverQueue[index].epochId = _epochId;
446:                    // only pay relayer for successful mints
447:                    executions++;
448:                }
449:            }
450:            index++;
451:        }
452:
...         // [...]
459: }
```

src/v2/VaultV2.previewWithdraw

```
357: function previewWithdraw(uint256 _id, uint256 _assets)
358:     public
359:     view
360:     override(SemiFungibleVault)
361:     returns (uint256 entitledAmount)
362: {
363:     // entitledAmount amount is derived from the claimTVL and the finalTVL
364:     // if user deposited 1000 assets and the claimTVL is 50% lower than
↪  finalTVL, the user is entitled to 500 assets
365:     // if user deposited 1000 assets and the claimTVL is 50% higher than
↪  finalTVL, the user is entitled to 1500 assets
366:     entitledAmount = _assets.mulDivDown(claimTVL[_id], finalTVL[_id]);
367: }
```

Tool used

Manual Review

Recommendation

Consider checking the total assets of the epoch `queue[index].epochId` to be greater than 0 before calling `previewWithdraw` in line 396.

Discussion

3xHarry

will move check from line `403` up before `previewWithdraw`, also considering implementing rollover delisting if `assetsToMint` is less than relayerFee

3xHarry

in general delisting of stale rollovers (not enough to pay for relayer, or not won prev epoch) should be delisted by smart contract.

Issue M-20: Arbitrum sequencer downtime lasting before and beyond epoch expiry prevents triggering depeg

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/422

Found by

0xnirlin, Respx, holyhansss, Dug, ShadowForce, libratus, spyrosonic10, berndartmueller, ltyu

Summary

A depeg event can not be triggered if the Arbitrum sequencer went down before the epoch ends and remains down beyond the epoch expiry. Instead, the collateral vault users can unfairly end the epoch without a depeg and claim the premium payments.

Vulnerability Detail

A depeg event can be triggered during an ongoing epoch by calling the `ControllerPeggedAssetV2.triggerDepeg` function. This function retrieves the latest price of the pegged asset via the `getLatestPrice` function.

If the Arbitrum sequencer is down or the grace period has not passed after the sequencer is back up, the `getLatestPrice` function reverts and the depeg event can not be triggered.

In case the sequencer went down before the epoch expired and remained down well after the epoch expired, a depeg can not be triggered, and instead, the epoch can be incorrectly ended without a depeg by calling the `ControllerPeggedAssetV2.triggerEndEpoch` function. Incorrectly, because at the time of the epoch expiry, it was not possible to trigger a depeg and hence it would be unfair to end the epoch without a depeg.

Impact

A depeg event can not be triggered, and premium vault users lose out on their insurance payout, while collateral vault users can wrongfully end the epoch and claim the premium.

Code Snippet

v2/Controllers/ControllerPeggedAssetV2.sol - triggerDepeg()

```
051: function triggerDepeg(uint256 _marketId, uint256 _epochId) public {
052:     address[2] memory vaults = vaultFactory.getVaults(_marketId);
```

```
053:
054:     if (vaults[0] == address(0) || vaults[1] == address(0))
055:         revert MarketDoesNotExist(_marketId);
056:
057:     IVaultV2 premiumVault = IVaultV2(vaults[0]);
058:     IVaultV2 collateralVault = IVaultV2(vaults[1]);
059:
060:     if (premiumVault.epochExists(_epochId) == false) revert EpochNotExist();
061:
062:     int256 price = getLatestPrice(premiumVault.token());
063:
064:     if (int256(premiumVault.strike()) <= price)
065:         revert PriceNotAtStrikePrice(price);
066:
...      // [...]
138: }
```

v2/Controllers/ControllerPeggedAssetV2.sol - getLatestPrice()

```
273: function getLatestPrice(address _token) public view returns (int256) {
274:     (
275:         ,
276:         /*uint80 roundId*/
277:         int256 answer,
278:         uint256 startedAt, /*uint256 updatedAt*/ /*uint80 answeredInRound*/
279:         ,
280:
281:     ) = sequencerUptimeFeed.latestRoundData();
282:
283:     // Answer == 0: Sequencer is up
284:     // Answer == 1: Sequencer is down
285:     bool isSequencerUp = answer == 0;
286:     if (!isSequencerUp) {
287:         revert SequencerDown();
288:     }
289:
290:     // Make sure the grace period has passed after the sequencer is back up.
291:     uint256 timeSinceUp = block.timestamp - startedAt;
292:     if (timeSinceUp <= GRACE_PERIOD_TIME) {
293:         revert GracePeriodNotOver();
294:     }
295:
...      // [...]
318: }
```

Tool used

Manual Review

Recommendation

Consider adding an additional "challenge" period (with reasonable length of time) after the epoch has expired and before the epoch end can be triggered without a depeg.

Within this challenge period, anyone can claim a depeg has happened during the epoch's expiry and trigger the epoch end. By providing the Chainlink round id's for both feeds (sequencer and price) at the time of the epoch expiry (`epochEnd`), the claim can be verified to assert that the sequencer was down and the strike price was reached.

Discussion

3xHarry

We are aware of this mechanic, however, users prefer to have the atomicity of instant settlement, this is so that users can utilize farming y2k tokens most effectively by rotating from one epoch to the next. Users are made aware of the risks when using chainlink oracles as well as the execution environment being on Arbitrum.

Issue M-21: VaultFactoryV2#changeTreasury misconfigures the vault

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/435

Found by

0xnirlin, 0x52, holyhansss, HonorLt, TrungOre, nobody2018, roguereddwarf, ni8mare, Dug, spyrosonic10, ast3ros, volodya, Ch_301, ElKu, warRoom

Summary

VaultFactoryV2#changeTreasury misconfigures the vault because the setTreasury subcall uses the wrong variable

Vulnerability Detail

VaultFactoryV2.sol#L228-L246

```
function changeTreasury(uint256 _marketId, address _treasury)
    public
    onlyTimeLocker
{
    if (_treasury == address(0)) revert AddressZero();

    address[2] memory vaults = marketIdToVaults[_marketId];

    if (vaults[0] == address(0) || vaults[1] == address(0)) {
        revert MarketDoesNotExist(_marketId);
    }

    IVaultV2(vaults[0]).whiteListAddress(_treasury);
    IVaultV2(vaults[1]).whiteListAddress(_treasury);
    IVaultV2(vaults[0]).setTreasury(treasury);
    IVaultV2(vaults[1]).setTreasury(treasury);

    emit AddressWhitelisted(_treasury, _marketId);
}
```

When setting the treasury for the underlying vault pair it accidentally use the treasury variable instead of _treasury. This means it uses the local VaultFactoryV2 treasury rather than the function input.

ControllerPeggedAssetV2.sol#L111-L123

```
premiumVault.sendTokens(_epochId, premiumFee, treasury);
premiumVault.sendTokens(
    _epochId,
    premiumTVL - premiumFee,
    address(collateralVault)
);
// strike price is reached so collateral is still entitled to premiumTVL -
 ↪  premiumFee but looses collateralTVL
collateralVault.sendTokens(_epochId, collateralFee, treasury);
collateralVault.sendTokens(
    _epochId,
    collateralTVL - collateralFee,
    address(premiumVault)
);
```

This misconfiguration can be damaging as it may cause the triggerDepeg call in the controller to fail due to the sendToken subcall. Additionally the time lock is the one required to call it which has a minimum of 3 days wait period. The result is that valid depegs may not get paid out since they are time sensitive.

Impact

Valid depegs may be missed due to misconfiguration

Code Snippet

ControllerPeggedAssetV2.sol#L51-L138

Tool used

Manual Review

Recommendation

Set to _treasury rather than treasury.

Discussion

3xHarry

good catch!

Issue M-22: Null epochs will freeze rollovers

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/442

Found by

bin2chen, 0x52, p0wd3r, berndartmueller, iglyx

Summary

When rolling a position it is required that the user didn't payout on the last epoch. The issue with the check is that if a null epoch is triggered then rollovers will break even though the vault didn't make a payout

Vulnerability Detail

Carousel.sol#L401-L406

```
uint256 entitledShares = previewWithdraw(
    queue[index].epochId,
    queue[index].assets
);
// mint only if user won epoch he is rolling over
if (entitledShares > queue[index].assets) {
```

When minting rollovers the following check is made so that the user won't automatically roll over if they made a payout last epoch. This check however will fail if there is ever a null epoch. Since no payout is made for a null epoch it should continue to rollover but doesn't.

Impact

Rollover will halt after null epoch

Code Snippet

Carousel.sol#L361-L459

Tool used

Manual Review

Recommendation

Change to less than or equal to:

```
-           if (entitledShares > queue[index].assets) {
+           if (entitledShares >= queue[index].assets) {
```

Discussion

3xHarry

makes sense

3xHarry

Won't be able to fix this edge case. Changes in the rollover queue make it now that positions are not deleted anymore but rather marked to 0 to prevent rollover queue manipulation. In this case, users would have to resolve their stuck rollover position manually. https://github.com/Y2K-Finance/Earthquake/pull/127

Issue M-23: Inconsistent use of epochBegin could lock user funds

Source: https://github.com/sherlock-audit/2023-03-Y2K-judging/issues/480

Found by

datapunk, toshii, sinarette, minhtrng, TrungOre, b4by_y0d4, evan, yixxas, berndartmueller, volodya, KingNFT

Summary

The epochBegin timestamp is used inconsistently and could lead to user funds being locked.

Vulnerability Detail

The function `ControllerPeggedAssetV2.triggerNullEpoch` checks for timestamp like this:

```
if (block.timestamp < uint256(epochStart)) revert EpochNotStarted();
```

The modifier `epochHasNotStarted` (used by `Carousel.deposit`) checks it like this:

```
if (block.timestamp > epochConfig[_id].epochBegin)
    revert EpochAlreadyStarted();
```

Both functions can be called when `block.timestamp == epochBegin`. This could lead to a scenario where a deposit happens after `triggerNullEpoch` is called (both in the same block). Because `triggerNullEpoch` sets the value for `finalTVL`, the TVL that comes from the deposit is not accounted for. If emissions have been distributed this epoch, this will lead to the incorrect distribution of emissions and once all emissions have been claimed the remaining assets will not be claimable, due to reversion in `withdraw` when trying to send emissions:

```
function previewEmissionsWithdraw(uint256 _id, uint256 _assets)
    public
    view
    returns (uint256 entitledAmount)
{
    entitledAmount = _assets.mulDivDown(emissions[_id], finalTVL[_id]);
}
...
//in withdraw:
uint256 entitledEmissions = previewEmissionsWithdraw(_id, _assets);
```

```
if (epochNull[_id] == false) {
    entitledShares = previewWithdraw(_id, _assets);
} else {
    entitledShares = _assets;
}
if (entitledShares > 0) {
    SemiFungibleVault.asset.safeTransfer(_receiver, entitledShares);
}
if (entitledEmissions > 0) {
    emissionsToken.safeTransfer(_receiver, entitledEmissions);
}
```

The above could also lead to revert through division by 0 if `finalTVL` is set to 0, even though the deposit after was successful.

Impact

incorrect distribution, Loss of deposited funds

Code Snippet

https://github.com/sherlock-audit/2023-03-Y2K/blob/ae7f210d8fbf21b9abf09ef30edfa548f7ae1aef/Earthquake/src/v2/VaultV2.sol#L433

Tool used

Manual Review

Recommendation

The modifier `epochHasNotStarted` should use >= as comparator