



SHERLOCK

SHERLOCK SECURITY REVIEW FOR



Prepared for:

Derby

Prepared by:

Sherlock

Lead Security Expert:

hyh

Dates Audited:

February 27 - March 13, 2023

Prepared on:

April 9, 2023

Introduction

Derby Finance is a community powered yield optimizer that diversifies its exposure over a wide variety of DeFi yield opportunities on different EVM chains and layer 2s.

Scope

[derby-yield-optimiser @ a20f134fd711dc418ed1a947431ded800a3ebace](#)

- [derby-yield-optimiser/contracts/Controller.sol](#)
- [derby-yield-optimiser/contracts/DerbyToken.sol](#)
- [derby-yield-optimiser/contracts/Game.sol](#)
- [derby-yield-optimiser/contracts/Interfaces/IController.sol](#)
- [derby-yield-optimiser/contracts/Interfaces/IGame.sol](#)
- [derby-yield-optimiser/contracts/Interfaces/IGoverned.sol](#)
- [derby-yield-optimiser/contracts/Interfaces/IProvider.sol](#)
- [derby-yield-optimiser/contracts/Interfaces/IVault.sol](#)
- [derby-yield-optimiser/contracts/Interfaces/IXChainController.sol](#)
- [derby-yield-optimiser/contracts/Interfaces/IXProvider.sol](#)
- [derby-yield-optimiser/contracts/MainVault.sol](#)
- [derby-yield-optimiser/contracts/Providers/AaveProvider.sol](#)
- [derby-yield-optimiser/contracts/Providers/BetaProvider.sol](#)
- [derby-yield-optimiser/contracts/Providers/CompoundProvider.sol](#)
- [derby-yield-optimiser/contracts/Providers/IdleProvider.sol](#)
- [derby-yield-optimiser/contracts/Providers/TruefiProvider.sol](#)
- [derby-yield-optimiser/contracts/Providers/YearnProvider.sol](#)
- [derby-yield-optimiser/contracts/TokenTimelock.sol](#)
- [derby-yield-optimiser/contracts/Vault.sol](#)
- [derby-yield-optimiser/contracts/VaultToken.sol](#)
- [derby-yield-optimiser/contracts/XChainController.sol](#)
- [derby-yield-optimiser/contracts/XProvider.sol](#)
- [derby-yield-optimiser/contracts/libraries/Swap.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
47	4

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

[hyh](#)
[rvierdiiev](#)
[Jeiwan](#)
[cergyk](#)
[Ch_301](#)
[c7e7eff](#)
[immeas](#)
[gogo](#)
[Met](#)
[spyrosonic10](#)
[PRAISE](#)
[oot2k](#)
[Sulpiride](#)
[foxb868](#)
[SPYBOY](#)
[nobody2018](#)

[Ruhum](#)
[Bauer](#)
[chaduke](#)
[saidam017](#)
[Bobface](#)
[XKET](#)
[sayan_](#)
[Auditwolf](#)
[evan](#)
[chainNue](#)
[csanuragjain](#)
[bin2chen](#)
[psy4n0n](#)
[Nyx](#)
[KingNFT](#)
[HonorLt](#)

[tallo](#)
[tives](#)
[atrixs](#)
[wzrdk3lly](#)
[ff](#)
[Saeedalipoor01988](#)
[SunSec](#)
[R-Nemes](#)
[tsvetanovv](#)
[koxuan](#)
[martin](#)
[Bnke0x0](#)
[gmx](#)
[Avci](#)
[8olidity](#)



Issue H-1: Anyone can execute certain functions that use cross chain messages and potentially cancel them with potential loss of funds.

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/390>

Found by

rvierdiiev, c7e7eff, Jeiwan

c7e7eff High

Summary

Certain functions that route messages cross chain on the Game and MainVault contract are unprotected (anyone can call them under the required state of the vaults). The way the cross chain messaging is implemented in the XProvider makes use of Connex's `xcall()` and sets the `msg.sender` as the `delegate` and `msg.value` as `relayerFee`. There are two possible attack vectors with this:

- Either an attacker can call the function and set the `msg.value` to low so it won't be relayed until someone bumps the fee (Connex allows anyone to bump the fee). This however means special action must be taken to bump the fee in such a case.
- Or the attacker can call the function (which irreversibly changes the state of the contract) and as the `delegate` of the `xcall` cancel the message. This functionality is however not yet active on Connex, but the moment it is the attacker will be able to change the state of the contract on the origin chain and make the cross chain message not execute on the destination chain leaving the contracts on the two chains out of synch with possible loss of funds as a result.

Vulnerability Detail

The XProvider contract's `xsend()` function sets the `msg.sender` as the `delegate` and `msg.value` as `relayerFee`

```
uint256 relayerFee = _relayerFee != 0 ? _relayerFee : msg.value;
IConnex(connex).xcall{value: relayerFee}(
    _destinationDomain, // _destination: Domain ID of the destination chain
    target, // _to: address of the target contract
    address(0), // _asset: use address zero for 0-value transfers
    msg.sender, // _delegate: address that can revert or forceLocal on
    ↪ destination
    0, // _amount: 0 because no funds are being transferred
```



```

    0, // _slippage: can be anything between 0-10000 because no funds are being
↳ transferred
    _callData // _callData: the encoded calldata to send
  );
}

```

xTransfer() using msg.sender as delegate:

```

IConnex(connex).xcall{value: (msg.value - _relayerFee)}(
  _destinationDomain, // _destination: Domain ID of the destination chain
  _recipient, // _to: address receiving the funds on the destination
  _token, // _asset: address of the token contract
  msg.sender, // _delegate: address that can revert or forceLocal on
↳ destination
  _amount, // _amount: amount of tokens to transfer
  _slippage, // _slippage: the maximum amount of slippage the user will accept
↳ in BPS (e.g. 30 = 0.3%)
  bytes("") // _callData: empty bytes because we're only sending funds
);
}

```

Connex [documentation](#) explaining:

```

params.delegate | (optional) Address allowed to cancel an xcall on destination.

```

Connex [documentation](#) seems to indicate this functionality isn't active yet though it isn't clear whether that applies to the cancel itself or only the bridging back the funds to the origin chain.

Impact

An attacker can call certain functions which leave the relying contracts on different chains in an unsynced state, with possible loss of funds as a result (mainly on XChainControleler's sendFundsToVault() when actual funds are transferred.

Code Snippet

MainVault's pushTotalUnderlyingToController() has no access control and calls pushTotalUnderlying() on the xProvider <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L249>

xProvider's pushTotalUnderlying() calling xsend() <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L243>

MainVault's sendRewardsToGame() has no access control and calls pushRewardsToGame() on the xProvider <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L365>



xProvider's `pushRewardsToGame()` calling `xsend()` <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L443>

XProvider setting `msg.sender` as delegate and `msg.value` as relayer fee:
<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L115-L124>

XChainController's unprotected `sendFundsToVault()`
<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L409-L414>

XProvider's `xTransfer()` setting `msg.sender` as delegate:
<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L152-L161>

Also on the Game contract, the unprotected `psuhAllocationsToController()` function: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L424>

and the `pushAllocationsToVaults()` <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L465>

Tool used

Manual Review

Recommendation

Provide access control limits to the functions sending message across Connex so only the Guardian can call these functions with the correct `msg.value` and do not use `msg.sender` as a delegate but rather a configurable address like the Guardian.



Issue H-2: CompoundProvider's balanceUnderlying and calcShares outputs are scaled incorrectly

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/202>

Found by

hyh, PRAISE, Jeiwan, foxb868

Summary

balanceUnderlying() returns values scaled with CToken decimals instead of the underlying decimals.

calcShares() also incorrectly scales its output, which have underlying decimals instead of the CToken decimals.

Vulnerability Detail

Underlying reason is exchangeRate() output is similarly and incorrectly processed in both functions. This results in wrong decimals of the both return values.

CToken and underlying decimals can differ, for example cDAI has 8 decimals, while DAI has 18.

Impact

Amount of the underlying held by Compound pool and amount of shares needed are misstated by magnitudes, which can lead to protocol-wide losses.

Code Snippet

balanceUnderlying() cancels out exchangeRate() decimals with dividing by $10^{(18 - 8 + \text{Underlying Token Decimals})}$, which results in CToken decimals:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/CompoundProvider.sol#L86-L100>

```
/// @notice Get balance from address in underlying token
/// @param _address Address to request balance from, most likely a Vault
/// @param _cToken Address of protocol LP Token eg cUSDC
/// @return balance in underlying token
function balanceUnderlying(
    address _address,
    address _cToken
) public view override returns (uint256) {
```



```

uint256 balanceShares = balance(_address, _cToken);
// The returned exchange rate from comp is scaled by 1 * 10^(18 - 8 +
↳ Underlying Token Decimals).
uint256 price = exchangeRate(_cToken);
uint256 decimals = IERC20Metadata(CToken(_cToken).underlying()).decimals();

return (balanceShares * price) / 10 ** (10 + decimals);
}

```

As balance() returns CToken balance and have CToken decimals:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/CompoundProvider.sol#L113-L120>

```

/// @notice Get balance of cToken from address
/// @param _address Address to request balance from
/// @param _cToken Address of protocol LP Token eg cUSDC
/// @return number of shares i.e LP tokens
function balance(address _address, address _cToken) public view override returns
↳ (uint256) {
    uint256 _balanceShares = IToken(_cToken).balanceOf(_address);
    return _balanceShares;
}

```

While exchangeRate() is CToken's exchangeRateStored() with (18 - 8 + Underlying Token Decimals) decimals:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/CompoundProvider.sol#L122-L129>

```

/// @notice Exchange rate of underlying protocol token
/// @dev returned price from compound is scaled
↳ https://compound.finance/docs/ctokens#exchange-rate
/// @param _cToken Address of protocol LP Token eg cUSDC
/// @return price of LP token
function exchangeRate(address _cToken) public view override returns (uint256) {
    uint256 _price = IToken(_cToken).exchangeRateStored();
    return _price;
}

```

<https://docs.compound.finance/v2/ctokens/#exchange-rate>

So balanceUnderlying() have CTokens decimals, which differ from underlying decimals, for example cUSDC have decimals of 8:

<https://etherscan.io/token/0x39aa39c021dfbae8fac545936693ac917d5e7563>

While USDC have decimals of 6:



<https://etherscan.io/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48>

cDAI has 8 decimals:

<https://etherscan.io/token/0x5d3a536e4d6dbd6114cc1ead35777bab948e3643>

DAI has 18:

<https://etherscan.io/token/0x6b175474e89094c44da98b954eedeac495271d0f>

balanceUnderlying() is used in Vault rebalancing:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L178-L192>

```
function rebalanceCheckProtocols(
    uint256 _newTotalUnderlying
) internal returns (uint256[] memory) {
    uint256[] memory protocolToDeposit = new
    ↪ uint[](controller.latestProtocolId(vaultNumber));
    uint256 latestID = controller.latestProtocolId(vaultNumber);
    for (uint i = 0; i < latestID; i++) {
        bool isBlacklisted = controller.getProtocolBlacklist(vaultNumber, i);

        storePriceAndRewards(_newTotalUnderlying, i);

        if (isBlacklisted) continue;
        setAllocation(i);

        int256 amountToProtocol = calcAmountToProtocol(_newTotalUnderlying, i);
        uint256 currentBalance = balanceUnderlying(i);
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L111-L118>

```
function pullFunds(uint256 _value) internal {
    uint256 latestID = controller.latestProtocolId(vaultNumber);
    for (uint i = 0; i < latestID; i++) {
        if (currentAllocations[i] == 0) continue;

        uint256 shortage = _value - vaultCurrency.balanceOf(address(this));
        uint256 balanceProtocol = balanceUnderlying(i);
```

Same for calcShares(), where shares scale is corrected wrongly, it has underlying decimals instead of CToken decimals:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/CompoundProvider.sol#L107-L111>



```
function calcShares(uint256 _amount, address _cToken) external view override
↳ returns (uint256) {
    uint256 decimals = IERC20Metadata(CToken(_cToken).underlying()).decimals();
    uint256 shares = (_amount * (10 ** (10 + decimals))) / exchangeRate(_cToken);
    return shares;
}
```

calcShares() is used in rebalancing as well:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L307-L315>

```
function withdrawFromProtocol(uint256 _protocolNum, uint256 _amount) internal {
    if (_amount <= 0) return;
    IController.ProtocolInfoS memory protocol = controller.getProtocolInfo(
        vaultNumber,
        _protocolNum
    );

    _amount = (_amount * protocol.uScale) / uScale;
    uint256 shares = IProvider(protocol.provider).calcShares(_amount,
↳ protocol.LPToken);
}
```

Tool used

Manual Review

Recommendation

Compound exchange rate is specifically scaled so that CToken_balance * price has 18 + Underlying Token Decimals, so it's enough to use 1e18:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/CompoundProvider.sol#L86-L100>

```
/// @notice Get balance from address in underlying token
/// @param _address Address to request balance from, most likely a Vault
/// @param _cToken Address of protocol LP Token eg cUSDC
/// @return balance in underlying token
function balanceUnderlying(
    address _address,
    address _cToken
) public view override returns (uint256) {
    uint256 balanceShares = balance(_address, _cToken);
    // The returned exchange rate from comp is scaled by 1 * 10^(18 - 8 +
↳ Underlying Token Decimals).
```



```

uint256 price = exchangeRate(_cToken);
uint256 decimals = IERC20Metadata(ICToken(_cToken).underlying()).decimals();

-   return (balanceShares * price) / 10 ** (10 + decimals);
+   return (balanceShares * price) / 1e18;
}

```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/CompoundProvider.sol#L102-L111>

```

/// @notice Calculates how many shares are equal to the amount
/// @dev returned price from compound is scaled
↪ https://compound.finance/docs/ctokens#exchange-rate
/// @param _amount Amount in underlying token e.g USDC
/// @param _cToken Address of protocol LP Token eg cUSDC
/// @return number of shares i.e LP tokens
function calcShares(uint256 _amount, address _cToken) external view override
↪ returns (uint256) {
-   uint256 decimals = IERC20Metadata(ICToken(_cToken).underlying()).decimals();
-   uint256 shares = (_amount * (10 ** (10 + decimals))) / exchangeRate(_cToken);
+   uint256 shares = _amount * 1e18 / exchangeRate(_cToken);
    return shares;
}

```

Discussion

sjoerdsommen

duplicate with #132



Issue H-3: Vault executes swaps without slippage protection

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/64>

Found by

Ruhum, Bauer, Nyx, immeas, nobody2018, hyh, cergyk, Bobface, Jeiwan

Summary

The vault executes swaps without slippage protection. That will cause a loss of funds because of sandwich attacks.

Vulnerability Detail

Both in `Vault.claimTokens()` and `MainVault.withdrawRewards()` swaps are executed through the Swaps library. It calculates the slippage parameters itself which doesn't work. Slippage calculations (min out) have to be calculated *outside* of the swap transaction. Otherwise, it uses the already modified pool values to calculate the min out value.

Impact

Swaps will be sandwiched causing a loss of funds for users who withdraw their rewards.

Code Snippet

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/libraries/Swap.sol#L60-L97>

Tool used

Manual Review

Recommendation

Slippage parameters should be included in the tx's calldata and passed to the Swap library.

Discussion

sjoerdsommen



duplicate with #157 and #48



Issue H-4: pushVaultAmounts can be called multiple times if in the right state

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/31>

Found by

cergyk

Summary

In XChainController, pushVaultAmounts can be called multiple times on same vault, calling sendXChainAmount and sending funds cross chain multiple times and breaking protocol accounting.

Vulnerability Detail

In XChainController, pushVaultAmounts:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L295-L324>

can be called by anybody and does not check it has already been called with current allocations, which means that it will execute its action every time it is called.

Impact

Protocol accounting is completely broken as a result, leading to funds being stuck on a chain due to being unaccounted for

Code Snippet

Tool used

Manual Review

Recommendation

Cache the allocations on which pushVaultAmount has already been called, so only the difference is forwarded cross chain, or find a way to check if a cross chain action has already been sent.



Issue M-1: Rebalancing breaks and can corrupt the accounting if amountToProtocol or amountToChain turn negative

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/408>

Found by

hyh, XKET, c7e7eff

Summary

Derby operates with deltas, but doesn't check that resulting amounts to be allocated are positive.

This allows for possible griefing attack permanently corrupting protocol accounting.

Vulnerability Detail

Explicit casting is used for the amounts that can be negative. In this case no reverting will happen, but the corresponding values be corrupted. In this case it cannot be reversed.

Impact

Attacker can manipulate protocol to enter corrupted accounting state by pushing such allocations that resulting protocol or chain allocations turn out to be negative, be converted to a `uint` and either block rebalancing or corrupt Derby state.

Code Snippet

`rebalanceCheckProtocols()` uses `uint(amountToProtocol)`:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L178-L203>

```
function rebalanceCheckProtocols(
    uint256 _newTotalUnderlying
) internal returns (uint256[] memory) {
    uint256[] memory protocolToDeposit = new
    ↪ uint[] (controller.latestProtocolId(vaultNumber));
    uint256 latestID = controller.latestProtocolId(vaultNumber);
    for (uint i = 0; i < latestID; i++) {
        bool isBlacklisted = controller.getProtocolBlacklist(vaultNumber, i);
```



```

storePriceAndRewards(_newTotalUnderlying, i);

if (isBlacklisted) continue;
setAllocation(i);

int256 amountToProtocol = calcAmountToProtocol(_newTotalUnderlying, i);
uint256 currentBalance = balanceUnderlying(i);

int256 amountToDeposit = amountToProtocol - int(currentBalance);
uint256 amountToWithdraw = amountToDeposit < 0 ? currentBalance -
↪ uint(amountToProtocol) : 0;

if (amountToDeposit > marginScale) protocolToDeposit[i] =
↪ uint256(amountToDeposit);
if (amountToWithdraw > uint(marginScale) || currentAllocations[i] == 0)
    withdrawFromProtocol(i, amountToWithdraw);
}

return protocolToDeposit;
}

```

calcAmountToProtocol() can return negative value for amountToProtocol if currentAllocations[_protocol] is negative:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L209-L218>

```

function calcAmountToProtocol(
    uint256 _totalUnderlying,
    uint256 _protocol
) internal view returns (int256 amountToProtocol) {
    if (totalAllocatedTokens == 0) amountToProtocol = 0;
    else
        amountToProtocol =
            (int(_totalUnderlying) * currentAllocations[_protocol]) /
            totalAllocatedTokens;
}

```

Which is renewed via setAllocation() by adding deltaAllocations:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L260-L264>

```

function setAllocation(uint256 _i) internal {
    currentAllocations[_i] += deltaAllocations[_i];
    deltaAllocations[_i] = 0;
    require(currentAllocations[_i] >= 0, "Allocation underflow");
}

```




```
}
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L178-L189>

```
function rebalanceCheckProtocols(
    uint256 _newTotalUnderlying
) internal returns (uint256[] memory) {
    uint256[] memory protocolToDeposit = new
    ↪ uint[](controller.latestProtocolId(vaultNumber));
    uint256 latestID = controller.latestProtocolId(vaultNumber);
    for (uint i = 0; i < latestID; i++) {
        ...
        setAllocation(i);
    }
}
```

deltaAllocations is set via receiveProtocolAllocationsInt() -> setDeltaAllocationsInt():

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L396-L400>

```
function setDeltaAllocationsInt(uint256 _protocolNum, int256 _allocation)
    ↪ internal {
    require(!controller.getProtocolBlacklist(vaultNumber, _protocolNum), "Protocol
    ↪ on blacklist");
    deltaAllocations[_protocolNum] += _allocation;
    deltaAllocatedTokens += _allocation;
}
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L350-L358>

```
function receiveProtocolAllocationsInt(int256[] memory _deltas) internal {
    for (uint i = 0; i < _deltas.length; i++) {
        int256 allocation = _deltas[i];
        if (allocation == 0) continue;
        setDeltaAllocationsInt(i, allocation);
    }

    deltaAllocationsReceived = true;
}
```

receiveProtocolAllocationsInt() is called by receiveProtocolAllocations() (onlyXProvider) and receiveProtocolAllocationsGuard() (onlyGuardian):

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L343-L345>



```
function receiveProtocolAllocations(int256[] memory _deltas) external
↳ onlyXProvider {
    receiveProtocolAllocationsInt(_deltas);
}
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L441-L443>

```
function receiveProtocolAllocationsGuard(int256[] memory _deltas) external
↳ onlyGuardian {
    receiveProtocolAllocationsInt(_deltas);
}
```

In normal workflow those are pushed from the game (step 6):

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L398-L425>

```
/// @notice Step 6 push; Game pushes deltaAllocations to vaults
/// @notice Push protocol allocation array from the game to all vaults/chains
/// @param _vault Address of the vault on given chainId
/// @param _deltas Array with delta allocations where the index matches the
↳ protocolId
function pushProtocolAllocationsToVault(
    uint32 _chainId,
    address _vault,
    int256[] memory _deltas
) external payable onlyGame {
    if (_chainId == homeChain) return
    ↳ IVault(_vault).receiveProtocolAllocations(_deltas);
    else {
        bytes4 selector =
        ↳ bytes4(keccak256("receiveProtocolAllocationsToVault(address,int256[])"));
        bytes memory callData = abi.encodeWithSelector(selector, _vault, _deltas);

        xSend(_chainId, callData, 0);
    }
}

/// @notice Step 6 receive; Game pushes deltaAllocations to vaults
/// @notice Receives protocol allocation array from the game to all vaults/chains
/// @param _vault Address of the vault on given chainId
/// @param _deltas Array with delta allocations where the index matches the
↳ protocolId
function receiveProtocolAllocationsToVault(
    address _vault,
```



```

    int256[] memory _deltas
) external onlySelf {
    return IVault(_vault).receiveProtocolAllocations(_deltas);
}

```

Similarly, `amountToChain` is `calcAmountToChain()` result:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L309-L320>

```

if (!getVaultChainIdOff(_vaultNumber, _chain)) {
    int256 amountToChain = calcAmountToChain(
        _vaultNumber,
        _chain,
        totalUnderlying,
        totalAllocation
    );
    (int256 amountToDeposit, uint256 amountToWithdraw) = calcDepositWithdraw(
        _vaultNumber,
        _chain,
        amountToChain
    );
}

```

`calcDepositWithdraw()` uses `uint256(_amountToChain)` without checks:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L330-L343>

```

function calcDepositWithdraw(
    uint256 _vaultNumber,
    uint32 _chainId,
    int256 _amountToChain
) internal view returns (int256, uint256) {
    uint256 currentUnderlying = getTotalUnderlyingOnChain(_vaultNumber, _chainId);

    int256 amountToDeposit = _amountToChain - int256(currentUnderlying);
    uint256 amountToWithdraw = amountToDeposit < 0
        ? currentUnderlying - uint256(_amountToChain)
        : 0;

    return (amountToDeposit, amountToWithdraw);
}

```

If `_amountToChain` be negative such explicit casting can yield big garbage unit:

<https://docs.soliditylang.org/en/latest/types.html#explicit-conversions>

This will most likely revert `pushVaultAmounts()` -> `calcDepositWithdraw()` call.



There is also a chance that `amountToWithdraw` computation completes, but the number end up being corrupted and will be used in `sendXChainAmount()`:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L368-L401>

```
function sendXChainAmount(
    uint256 _vaultNumber,
    uint32 _chainId,
    int256 _amountDeposit,
    uint256 _amountToWithdraw,
    uint256 _exchangeRate
) internal {
    address vault = getVaultAddress(_vaultNumber, _chainId);
    bool receivingFunds;
    uint256 amountToSend = 0;

    if (_amountDeposit > 0 && _amountDeposit < minimumAmount) {
        vaultStage[_vaultNumber].fundsReceived++;
    } else if (_amountDeposit >= minimumAmount) {
        receivingFunds = true;
        setAmountToDeposit(_vaultNumber, _chainId, _amountDeposit);
        vaultStage[_vaultNumber].fundsReceived++;
    }

    if (_amountToWithdraw > 0 && _amountToWithdraw < uint(minimumAmount)) {
        vaultStage[_vaultNumber].fundsReceived++;
    } else if (_amountToWithdraw >= uint(minimumAmount)) {
        amountToSend = _amountToWithdraw;
    }

    xProvider.pushSetXChainAllocation{value: msg.value}(
        vault,
        _chainId,
        amountToSend,
        _exchangeRate,
        receivingFunds
    );
    emit SendXChainAmount(vault, _chainId, amountToSend, _exchangeRate,
    ↪ receivingFunds);
}
```

Tool used

Manual Review



Recommendation

Consider controlling both values to be positive. It's needed to be done way before conversion itself, on user entry or processing stage.



Issue M-2: Native funds sent with pushVaultAmounts and sendFundsToVault can be lost

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/398>

Found by

hyh

Summary

XChainController's pushVaultAmounts() and sendFundsToVault() do not always use the native funds provided to cover the fees.

In the cases when the funds aren't used they are not returned, ending up frozen on the contract balance as no balance utilization is now implemented.

Vulnerability Detail

Native funds attached to the pushVaultAmounts() and sendFundsToVault() calls are frozen with XChainController when not used.

Impact

Caller's funds meant to cover x-chain transfers are permanently frozen on the XChainController balance when getVaultChainIdOff(_vaultNumber, _chain).

Code Snippet

pushVaultAmounts() doesn't call sendXChainAmount() and use funds if getVaultChainIdOff(_vaultNumber, _chain):

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L295-L324>

```
function pushVaultAmounts(
    uint256 _vaultNumber,
    uint16 _chain
) external payable onlyWhenUnderlyingsReceived(_vaultNumber) {
    address vault = getVaultAddress(_vaultNumber, _chain);
    require(vault != address(0), "xChainController: not a valid vaultnumber");
    int256 totalAllocation = getCurrentTotalAllocation(_vaultNumber);
    uint256 totalWithdrawalRequests = getTotalWithdrawalRequests(_vaultNumber);
    uint256 totalUnderlying = getTotalUnderlyingVault(_vaultNumber) -
    ↪ totalWithdrawalRequests;
    uint256 totalSupply = getTotalSupply(_vaultNumber);
```



```

uint256 decimals = xProvider.getDecimals(vault);
uint256 newExchangeRate = (totalUnderlying * (10 ** decimals)) / totalSupply;

>> if (!getVaultChainIdOff(_vaultNumber, _chain)) {
    int256 amountToChain = calcAmountToChain(
        _vaultNumber,
        _chain,
        totalUnderlying,
        totalAllocation
    );
    (int256 amountToDeposit, uint256 amountToWithdraw) = calcDepositWithdraw(
        _vaultNumber,
        _chain,
        amountToChain
    );

>>     sendXChainAmount(_vaultNumber, _chain, amountToDeposit, amountToWithdraw,
↪     newExchangeRate);
    }
}

```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L368-L401>

```

function sendXChainAmount(
    uint256 _vaultNumber,
    uint32 _chainId,
    int256 _amountDeposit,
    uint256 _amountToWithdraw,
    uint256 _exchangeRate
) internal {
    address vault = getVaultAddress(_vaultNumber, _chainId);
    bool receivingFunds;
    uint256 amountToSend = 0;

    if (_amountDeposit > 0 && _amountDeposit < minimumAmount) {
        vaultStage[_vaultNumber].fundsReceived++;
    } else if (_amountDeposit >= minimumAmount) {
        receivingFunds = true;
        setAmountToDeposit(_vaultNumber, _chainId, _amountDeposit);
        vaultStage[_vaultNumber].fundsReceived++;
    }

    if (_amountToWithdraw > 0 && _amountToWithdraw < uint(minimumAmount)) {
        vaultStage[_vaultNumber].fundsReceived++;
    } else if (_amountToWithdraw >= uint(minimumAmount)) {

```



```

        amountToSend = _amountToWithdraw;
    }

>> xProvider.pushSetXChainAllocation{value: msg.value}(
    vault,
    _chainId,
    amountToSend,
    _exchangeRate,
    receivingFunds
);
emit SendXChainAmount(vault, _chainId, amountToSend, _exchangeRate,
↪ receivingFunds);
}

```

sendFundsToVault() similarly do not use funds when
getVaultChainIdOff(_vaultNumber, _chain):

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L409-L441>

```

function sendFundsToVault(
    uint256 _vaultNumber,
    uint256 _slippage,
    uint32 _chain,
    uint256 _relayerFee
) external payable onlyWhenFundsReceived(_vaultNumber) {
    address vault = getVaultAddress(_vaultNumber, _chain);
    require(vault != address(0), "xChainController: not a valid vaultnumber");
>> if (!getVaultChainIdOff(_vaultNumber, _chain)) {
    uint256 amountToDeposit = getAmountToDeposit(_vaultNumber, _chain);

    if (amountToDeposit > 0) {
        address underlying = getUnderlyingAddress(_vaultNumber, _chain);

        uint256 balance = IERC20(underlying).balanceOf(address(this));
        if (amountToDeposit > balance) amountToDeposit = balance;

        IERC20(underlying).safeIncreaseAllowance(address(xProvider),
↪ amountToDeposit);
>> xProvider.xTransferToVaults{value: msg.value}(
    vault,
    _chain,
    amountToDeposit,
    underlying,
    _slippage,
    _relayerFee
);
    setAmountToDeposit(_vaultNumber, _chain, 0);
}

```




```
        emit SentFundsToVault(vault, _chain, amountToDeposit, underlying);
    }
}
vaultStage[_vaultNumber].fundsSent++;
if (vaultStage[_vaultNumber].fundsSent == chainIds.length)
↪ resetVaultStages(_vaultNumber);
}
```

Tool used

Manual Review

Recommendation

Consider returning the funds to the caller in both cases.

```
if (!getVaultChainIdOff(_vaultNumber, _chain)) {
    ...
} else {
    if (msg.value > 0) {
        send msg.value to the caller
    }
}
```

Discussion

sjoerdsommen

payable eth amount to cover connext fees are stuck in the xchaincontroller when the funds are sent to the same chain as the xchaincontroller; correct but as we do the triggers ourselves we would never send any funds when doing the trigger to the homechain. Therefore not high but medium.



Issue M-3: withdrawal request override

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/392>

Found by

XKET, oot2k, bin2chen, immeas, nobody2018, gogo, saidam017, HonorLt

Summary

It is possible that a withdrawal request is overridden during the initial phase.

Vulnerability Detail

Users have two options to withdraw: directly or request a withdrawal if not enough funds are available at the moment.

When making a `withdrawalRequest` it is required that the user has `withdrawalRequestPeriod` not set:

```
function withdrawalRequest(
  uint256 _amount
) external nonReentrant onlyWhenVaultIsOn returns (uint256 value) {
  UserInfo storage user = userInfo[msg.sender];
  require(user.withdrawalRequestPeriod == 0, "Already a request");

  value = (_amount * exchangeRate) / (10 ** decimals());

  _burn(msg.sender, _amount);

  user.withdrawalAllowance = value;
  user.withdrawalRequestPeriod = rebalancingPeriod;
  totalWithdrawalRequests += value;
}
```

This will misbehave during the initial period when `rebalancingPeriod` is 0. The check will pass, so if invoked multiple times, it will burn users' shares and overwrite the value.

Impact

While not very likely to happen, the impact would be huge, because the users who invoke this function several times before the first rebalance, would burn their shares and lose previous `withdrawalAllowance`. The protocol should prevent such mistakes.



Code Snippet

I have extended one of your test cases to showcase this vulnerability:

```
it('Should not be able to lose previous withdrawal request :(', async function
↳ () {
  const { vault, user } = await setupVault();
  await vault.connect(user).deposit(parseUSDC(10_000), user.address); // 10k
  expect(await vault.totalSupply()).to.be.equal(parseUSDC(10_000)); // 10k

  // mocking exchangerate to 0.9
  await vault.setExchangeRateTEST(parseUSDC(0.9));

  // withdrawal request for 2x 5k LP tokens
  await expect(() =>
    vault.connect(user).withdrawalRequest(parseUSDC(5_000)),
  ).to.changeTokenBalance(vault, user, -parseUSDC(5_000));

  await expect(() =>
    vault.connect(user).withdrawalRequest(parseUSDC(5_000)),
  ).to.changeTokenBalance(vault, user, -parseUSDC(5_000));

  // check withdrawalAllowance user and totalsupply
  expect(await
↳ vault.connect(user).getWithdrawalAllowance()).to.be.equal(parseUSDC(4_500));
  expect(await vault.totalSupply()).to.be.equal(parseUSDC(0));

  // trying to withdraw allowance before the vault reserved the funds
  await expect(vault.connect(user).withdrawAllowance()).to.be.revertedWith('');

  // mocking vault settings
  await vault.upRebalancingPeriodTEST();
  await vault.setReservedFundsTEST(parseUSDC(10_000));

  // withdraw allowance should give 4.5k USDC
  await expect(() =>
↳ vault.connect(user).withdrawAllowance()).to.changeTokenBalance(
    IUSDC,
    user,
    parseUSDC(4_500),
  );

  // trying to withdraw allowance again
  await expect(vault.connect(user).withdrawAllowance()).to.be.revertedWith('!All
↳ owance');
  });
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser>



/contracts/MainVault.sol#L153

Tool used

Manual Review

Recommendation

Require `rebalancingPeriod != 0` in `withdrawalRequest`, otherwise, force users to directly withdraw.



Issue M-4: Current period profit can be extracted from the Vault by front running state change before exchange rate recalculation

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/369>

Found by

hyh, Bobface, PRAISE

Summary

Every period `exchangeRate` recalculation can be estimated in advance and this can be used to extract funds from the Vault by depositing right before rebalances with big enough positive effects, earning this period profit without taking the risk.

Vulnerability Detail

`exchangeRate` can be estimated based on the performance of the underlying pools, all of which are public.

Since `exchangeRate` is recalculated and stored during rebalancing only and old rate is used for deposits until then, it's possible to extract value from the Vault.

Bob the attacker can monitor P&L of the Vault's pools invested and, whenever cumulative realized profit exceeds transaction expenses threshold, do deposit right before rebalancing while `State.Idle` and withdraw the funds right after it. Bob might want to front-run Vault State change or just deposit on observing the accumulated profit.

Even if this be paired with some delays, it is still be low risk / big reward operation for Bob as profit is guaranteed, while risk can be controllable (he can limit the attack in size to be able to withdraw immediately) and aren't high overall (say having earned free 10% on his funds he will need to wait until next rebalancing in order to withdraw; most probably this 10% will cover the associated losses of being invested for one rebalancing period, if any).

POC steps for Bob are:

- 1) Monitor estimated P&L of rebalancing
- 2) If big enough Deposit just before rebalancing
- 3) Withdraw immediately after, if possible, otherwise exit on the next rebalancing



Impact

It is free profit for the attacker at the expense of depositors, whose already realized profit can be heavily diluted this way.

Code Snippet

deposit() before rebalance with current exchangeRate() (which is factually outdated, but used):

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L101-L124>

```
/// @notice Deposit in Vault
/// @dev Deposit VaultCurrency to Vault and mint LP tokens
/// @param _amount Amount to deposit
/// @param _receiver Receiving adress for the tokens
/// @return shares Tokens received by buyer
function deposit(
    uint256 _amount,
    address _receiver
) external nonReentrant onlyWhenVaultIsOn returns (uint256 shares) {
    if (training) {
        require(whitelist[msg.sender]);
        uint256 balanceSender = (balanceOf(msg.sender) * exchangeRate) / (10 **
    ↪ decimals());
        require(_amount + balanceSender <= maxTrainingDeposit);
    }

    uint256 balanceBefore = getVaultBalance() - reservedFunds;
    vaultCurrency.safeTransferFrom(msg.sender, address(this), _amount);
    uint256 balanceAfter = getVaultBalance() - reservedFunds;

    uint256 amount = balanceAfter - balanceBefore;
    shares = (amount * (10 ** decimals())) / exchangeRate;

    _mint(_receiver, shares);
}
```

withdraw() right afterwards rebalancing with new exchangeRate() that is increased by the realized profit:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L126-L144>

```
/// @notice Withdraw from Vault
/// @dev Withdraw VaultCurrency from Vault and burn LP tokens
/// @param _amount Amount to withdraw in LP tokens
```



```

/// @param _receiver Receiving adress for the vaultcurrency
/// @return value Amount received by seller in vaultCurrency
function withdraw(
    uint256 _amount,
    address _receiver,
    address _owner
) external nonReentrant onlyWhenVaultIsOn returns (uint256 value) {
    value = (_amount * exchangeRate) / (10 ** decimals());

    require(value > 0, "!value");

    require(getVaultBalance() - reservedFunds >= value, "!funds");

    _burn(msg.sender, _amount);
    transferFunds(_receiver, value);
}

```

New exchangeRate is pushed during rebalancing either by XProvider or Guardian:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L285-L301>

```

/// @notice Step 3 end; xChainController pushes exchangeRate and amount the
↳ vaults have to send back to all vaults
/// @notice Will set the amount to send back to the xController by the
↳ xController
/// @dev Sets the amount and state so the dao can trigger the rebalanceXChain
↳ function
/// @dev When amount == 0 the vault doesnt need to send anything and will wait
↳ for funds from the xController
/// @param _amountToSend amount to send in vaultCurrency
function setXChainAllocationInt(
    uint256 _amountToSend,
    uint256 _exchangeRate,
    bool _receivingFunds
) internal {
    amountToSendXChain = _amountToSend;
    exchangeRate = _exchangeRate;

    if (_amountToSend == 0 && !_receivingFunds) settleReservedFunds();
    else if (_amountToSend == 0 && _receivingFunds) state = State.WaitingForFunds;
    else state = State.SendingFundsXChain;
}

```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L275-L283>



```
/// @notice See setXChainAllocationInt below
function setXChainAllocation(
    uint256 _amountToSend,
    uint256 _exchangeRate,
    bool _receivingFunds
) external onlyXProvider {
    require(state == State.PushedUnderlying, stateError);
    setXChainAllocationInt(_amountToSend, _exchangeRate, _receivingFunds);
}
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L426-L433>

```
/// @notice Step 3: Guardian function
function setXChainAllocationGuard(
    uint256 _amountToSend,
    uint256 _exchangeRate,
    bool _receivingFunds
) external onlyGuardian {
    setXChainAllocationInt(_amountToSend, _exchangeRate, _receivingFunds);
}
```

Tool used

Manual Review

Recommendation

Consider tying the deposits and withdrawals to the next period `exchangeRate`, i.e. gather the requests for both during current period, while processing them all with the updated exchange rate on the next rebalance. Rebalancing period can be somewhat lowered (say to 1 week initially, then possibly to 3-5 days) to allow for quicker funds turnaround.

Discussion

sjoerdsommen

Duplicate with #173 . We don't yet know how to fix it.



Issue M-5: Vault's savedTotalUnderlying tracks withdrawn funds incorrectly

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/363>

Found by

chaduke, spyrosonic10, rvierdiev, atrixs, Ch_301, hyh

Summary

Vault's pullFunds() updates total underlying funds outside Vault variable, savedTotalUnderlying, preliminary with the amount requested, not with the amount obtained.

Vulnerability Detail

pullFunds() reduces savedTotalUnderlying before calling for protocol withdrawal, with amount requested for withdrawal, not with amount that was actually withdrawn. Also, when amountToWithdraw < minimumPull the minimumPull is removed from savedTotalUnderlying despite no withdrawal is made in this case.

As amount requested tends to be bigger than amount withdrawn, the result is savedTotalUnderlying being understated this way. This effect will accumulate over time.

Impact

Net impact is ongoing Vault misbalancing by artificially reducing allocations to the underlying DeFi pools as understated savedTotalUnderlying means less funds are deemed to be available for each Provider.

This will reduce the effective investment sizes vs desired and diminish the results realized as some funds will remain systemically dormant (not used as calcUnderlyingIncBalance() is reported lower than real).

This is the protocol wide loss for the depositors.

Code Snippet

savedTotalUnderlying -= amountToWithdraw is performed before withdrawFromProtocol() was called:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L107-L127>



```

    /// @notice Withdraw from protocols on shortage in Vault
    /// @dev Keeps on withdrawing until the Vault balance > _value
    /// @param _value The total value of vaultCurrency an user is trying to
    ↪ withdraw.
    /// @param _value The (value - current underlying value of this vault) is
    ↪ withdrawn from the underlying protocols.
    function pullFunds(uint256 _value) internal {
        uint256 latestID = controller.latestProtocolId(vaultNumber);
        for (uint i = 0; i < latestID; i++) {
            if (currentAllocations[i] == 0) continue;

            uint256 shortage = _value - vaultCurrency.balanceOf(address(this));
            uint256 balanceProtocol = balanceUnderlying(i);

            uint256 amountToWithdraw = shortage > balanceProtocol ? balanceProtocol :
    ↪ shortage;
    >> savedTotalUnderlying -= amountToWithdraw;

            if (amountToWithdraw < minimumPull) break;
            withdrawFromProtocol(i, amountToWithdraw);

            if (_value <= vaultCurrency.balanceOf(address(this))) break;
        }
    }

```

But actual amount withdrawn from protocol differs from the amount requested:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L307-L336>

```

    function withdrawFromProtocol(uint256 _protocolNum, uint256 _amount) internal {
        if (_amount <= 0) return;
        IController.ProtocolInfoS memory protocol = controller.getProtocolInfo(
            vaultNumber,
            _protocolNum
        );

        _amount = (_amount * protocol.uScale) / uScale;
        uint256 shares = IProvider(protocol.provider).calcShares(_amount,
    ↪ protocol.LPToken);
        uint256 balance = IProvider(protocol.provider).balance(address(this),
    ↪ protocol.LPToken);

        if (shares == 0) return;
    >> if (balance < shares) shares = balance;

        IERC20(protocol.LPToken).safeIncreaseAllowance(protocol.provider, shares);
    }

```



```

>> uint256 amountReceived = IProvider(protocol.provider).withdraw(
    shares,
    protocol.LPToken,
    protocol.underlying
);

    if (protocol.underlying != address(vaultCurrency)) {
>>     _amount = Swap.swapStableCoins(
        Swap.SwapInOut(amountReceived, protocol.underlying,
↪ address(vaultCurrency)),
        controller.underlyingUScale(protocol.underlying),
        uScale,
        controller.getCurveParams(protocol.underlying, address(vaultCurrency))
    );
    }
}

```

This happens because of:

- 1) shares balance restriction
- 2) underlying funds availability within Provider
- 3) swapping to vaultCurrency is market dependent and is not deterministic

As this is substantial degree of randomness it can be assumed that `withdrawFromProtocol()` typically returns somewhat different amount than it was requested.

`savedTotalUnderlying` is then used by `calcUnderlyingIncBalance()`:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L156-L164>

```

    /// @notice Helper to return underlying balance plus totalUnderlying -
↪ liquidity for the vault
    /// @return underlying totalUnderlying - liquidityVault
    function calcUnderlyingIncBalance() internal view returns (uint256) {
>>     uint256 totalUnderlyingInclVaultBalance = savedTotalUnderlying +
        getVaultBalance() -
        reservedFunds;
        uint256 liquidityVault = (totalUnderlyingInclVaultBalance * liquidityPerc) /
↪ 100;
        return totalUnderlyingInclVaultBalance - liquidityVault;
    }

```

Which defines the allocation during Vault rebalancing:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L135-L154>



```

function rebalance() external nonReentrant {
    require(state == State.RebalanceVault, stateError);
    require(deltaAllocationsReceived, "!Delta allocations");

    rebalancingPeriod++;

    claimTokens();
    settleDeltaAllocation();

>> uint256 underlyingIncBalance = calcUnderlyingIncBalance();
    uint256[] memory protocolToDeposit =
↳ rebalanceCheckProtocols(underlyingIncBalance);

    executeDeposits(protocolToDeposit);
    setTotalUnderlying();

    if (reservedFunds > vaultCurrency.balanceOf(address(this)))
↳ pullFunds(reservedFunds);

    state = State.SendRewardsPerToken;
    deltaAllocationsReceived = false;
}

```

Tool used

Manual Review

Recommendation

Consider returning the amount realized in withdrawFromProtocol():

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L303-L336>

```

/// @notice Withdraw amount from underlying protocol
/// @dev shares = amount / PricePerShare
/// @param _protocolNum Protocol number linked to an underlying protocol e.g
↳ compound_usdc_01
/// @param _amount in VaultCurrency to withdraw
- function withdrawFromProtocol(uint256 _protocolNum, uint256 _amount) internal {
+ function withdrawFromProtocol(uint256 _protocolNum, uint256 _amount) internal
↳ returns (uint256 amountReceived) {
    if (_amount <= 0) return;
    IController.ProtocolInfoS memory protocol = controller.getProtocolInfo(
        vaultNumber,
        _protocolNum
    );
    uint256 amountRealized = protocol.getPricePerShare() * _amount;
    uint256 amountReceived = protocol.withdraw(_amount);
    return amountRealized;
}

```



```

    );

    _amount = (_amount * protocol.uScale) / uScale;
    uint256 shares = IProvider(protocol.provider).calcShares(_amount,
↳ protocol.LPToken);
    uint256 balance = IProvider(protocol.provider).balance(address(this),
↳ protocol.LPToken);

    if (shares == 0) return;
    if (balance < shares) shares = balance;

    IERC20(protocol.LPToken).safeIncreaseAllowance(protocol.provider, shares);
-   uint256 amountReceived = IProvider(protocol.provider).withdraw(
+   amountReceived = IProvider(protocol.provider).withdraw(
        shares,
        protocol.LPToken,
        protocol.underlying
    );

    if (protocol.underlying != address(vaultCurrency)) {
-   _amount = Swap.swapStableCoins(
+   amountReceived = Swap.swapStableCoins(
        Swap.SwapInOut(amountReceived, protocol.underlying,
↳ address(vaultCurrency)),
        controller.underlyingUScale(protocol.underlying),
        uScale,
        controller.getCurveParams(protocol.underlying, address(vaultCurrency))
    );
    }
}

```

And updating savedTotalUnderlying after the fact with the actual amount in pullFunds():

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L107-L127>

```

    /// @notice Withdraw from protocols on shortage in Vault
    /// @dev Keeps on withdrawing until the Vault balance > _value
    /// @param _value The total value of vaultCurrency an user is trying to
↳ withdraw.
    /// @param _value The (value - current underlying value of this vault) is
↳ withdrawn from the underlying protocols.
    function pullFunds(uint256 _value) internal {
        uint256 latestID = controller.latestProtocolId(vaultNumber);
        for (uint i = 0; i < latestID; i++) {
            if (currentAllocations[i] == 0) continue;

```



```

        uint256 shortage = _value - vaultCurrency.balanceOf(address(this));
        uint256 balanceProtocol = balanceUnderlying(i);

        uint256 amountToWithdraw = shortage > balanceProtocol ? balanceProtocol :
↪ shortage;
-     savedTotalUnderlying -= amountToWithdraw;

        if (amountToWithdraw < minimumPull) break;
-     withdrawFromProtocol(i, amountToWithdraw);
+     uint256 amountWithdrawn = withdrawFromProtocol(i, amountToWithdraw);
+     savedTotalUnderlying -= amountWithdrawn;

        if (_value <= vaultCurrency.balanceOf(address(this))) break;
    }
}

```

This will also remove `minimumPull` bias that is currently added when `if (amountToWithdraw < minimumPull) break` triggers (no funds are pulled, but `savedTotalUnderlying` is reduced by `minimumPull`).

Discussion

sjoerdsommen

`pullFunds` can diverge a bit between actual and calculated value, this causes `safedTotalUnderlying` to be incorrect; correct but not severe because small values and after rebalancing the issue does not persist

hrishibhat

Given the necessary preconditions required for the understated `savedTotalUnderlying`. Considering this a valid medium



Issue M-6: YearnProvider freezes yearn tokens on partial withdrawal

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/355>

Found by

hyh, spyrosonic10

Summary

YearnProvider's `withdraw()` doesn't account for partial withdrawal situation, which isn't rare, so the unused part of user shares end up being stuck on the contract balance as there is no mechanics to retrieve them thereafter.

Vulnerability Detail

Full amount of the shares user requested to be burned is transferred to YearnProvider, but only part of it can be utilized by Yearn withdrawal.

Liquidity shortage (squeeze) is common enough situation, for example it can occur whenever part of the Yearn strategy is tied to a lending market that have high utilization at the moment of the call.

Impact

Part of protocol funds can be permanently frozen on YearnProvider contract balance (as it's not operating the funds itself, always referencing the caller Vault).

As Provider's `withdraw` is routinely called by Vault managing the aggregated funds distribution, the freeze amount can be massive enough and will be translated to a loss for many users.

Code Snippet

`withdraw()` transfers all the requested `_amount` from the user, but do not return the remainder `_yToken` amount:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/YearnProvider.sol#L44-L66>

```
function withdraw(
    uint256 _amount,
    address _yToken,
    address _uToken
) external override returns (uint256) {
```



```

uint256 balanceBefore = IERC20(_uToken).balanceOf(msg.sender);

require(
    IYearn(_yToken).transferFrom(msg.sender, address(this), _amount) == true,
    "Error transferFrom"
);

uint256 uAmountReceived = IYearn(_yToken).withdraw(_amount);
IERC20(_uToken).safeTransfer(msg.sender, uAmountReceived);

uint256 balanceAfter = IERC20(_uToken).balanceOf(msg.sender);
require(
    (balanceAfter - balanceBefore - uAmountReceived) == 0,
    "Error Withdraw: under/overflow"
);

return uAmountReceived;
}

```

uAmountReceived can correspond only to a part of shares `_amount` obtained from the caller.

Yearn withdrawal is not guaranteed to be full, value returned and shares burned depend on availability (i.e. `shares < maxShares` is valid case):

<https://github.com/yearn/yearn-vaults/blob/master/contracts/Vault.vy#L1144-L1167>

The remaining part of the shares, `maxShares - shares`, end up left on the YearnProvider balance.

Provider's `withdraw()` is used by the Vault, where `amountReceived` is deemed corresponding to the full shares spent:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L303-L326>

```

/// @notice Withdraw amount from underlying protocol
/// @dev shares = amount / PricePerShare
/// @param _protocolNum Protocol number linked to an underlying protocol e.g
    ↪ compound_usdc_01
/// @param _amount in VaultCurrency to withdraw
function withdrawFromProtocol(uint256 _protocolNum, uint256 _amount) internal {
    if (_amount <= 0) return;
    IController.ProtocolInfoS memory protocol = controller.getProtocolInfo(
        vaultNumber,
        _protocolNum
    );
}

```




```

    _amount = (_amount * protocol.uScale) / uScale;
    uint256 shares = IProvider(protocol.provider).calcShares(_amount,
↳   protocol.LPToken);
    uint256 balance = IProvider(protocol.provider).balance(address(this),
↳   protocol.LPToken);

    if (shares == 0) return;
    if (balance < shares) shares = balance;

    IERC20(protocol.LPToken).safeIncreaseAllowance(protocol.provider, shares);
    uint256 amountReceived = IProvider(protocol.provider).withdraw(
        shares,
        protocol.LPToken,
        protocol.underlying
    );

```

Tool used

Manual Review

Recommendation

Consider returning the unused shares to the caller:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/YearnProvider.sol#L44-L66>

```

function withdraw(
    uint256 _amount,
    address _yToken,
    address _uToken
) external override returns (uint256) {
    uint256 balanceBefore = IERC20(_uToken).balanceOf(msg.sender);
+   uint256 sharesBefore = IERC20(_yToken).balanceOf(address(this));

    require(
        IYearn(_yToken).transferFrom(msg.sender, address(this), _amount) == true,
        "Error transferFrom"
    );

    uint256 uAmountReceived = IYearn(_yToken).withdraw(_amount);
    IERC20(_uToken).safeTransfer(msg.sender, uAmountReceived);

    uint256 balanceAfter = IERC20(_uToken).balanceOf(msg.sender);
    require(
        (balanceAfter - balanceBefore - uAmountReceived) == 0,
        "Error Withdraw: under/overflow"
    );
}

```



```
    );  
+   uint256 sharesAfter = IERC20(_yToken).balanceOf(address(this));  
+   if (sharesAfter > sharesBefore) {  
+       IERC20(_yToken).safeTransfer(msg.sender, sharesAfter - sharesBefore);  
+   }  
  
    return uAmountReceived;  
}
```



Issue M-7: Wrong type casting leads to unsigned integer underflow exception when current price is < last price

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/334>

Found by

sayan_, gogo

Summary

When the current price of a locked token is lower than the last price, the Vault.storePriceAndRewards will revert because of the wrong integer casting.

Vulnerability Detail

The following line appears in Vault.storePriceAndRewards:

```
int256 priceDiff = int256(currentPrice - lastPrices[_protocolId]);
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L233>

If lastPrices[_protocolId] is higher than the currentPrice, the solidity compiler will revert due the underflow of subtracting unsigned integers because it will first try to calculate the result of currentPrice - lastPrices[_protocolId] and **then** try to cast it to int256.

Impact

The rebalance will fail when the current token price is less than the last one stored.

Code Snippet

```
int256 priceDiff = int256(currentPrice - lastPrices[_protocolId]);
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L233>

Tool used

Manual Review



Recommendation

Casting should be performed in the following way to avoid underflow and to allow the priceDiff being negative:

```
int256 priceDiff = int256(currentPrice) - int256(lastPrices[_protocolId]);
```



Issue M-8: Rebalancing can be blocked when pulling funds from a TrueFi or a Idle vault

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/327>

Found by

c7e7eff, Jeiwan

Summary

An attempt to pull funds from a TrueFi or a Idle vault may revert and cause reverting of the `Vault.rebalance` function. To unlock the function, the team will need to top up the balance of the vault so that there's enough tokens to cover the reserved funds.

Vulnerability Detail

At the end of a rebalancing, the `Vault.rebalance` function is called to deposit and/or withdraw funds from protocol according to the allocations set by gamers. The function:

1. calculates vault's current balance;
2. computes the amounts to deposit or withdraw and withdraws excessive amounts from protocols;
3. deposits amounts to protocol;
4. ensures that there's enough tokens to cover the reserved funds and pulls funds from protocols when it's not enough.

Notice the order in which the function interacts with the protocols:

1. it withdraws excessive funds;
2. it deposits funds;
3. it withdraws funds when the vault doesn't have enough funds to cover the reserved funds.

However, TrueFi and Idle vaults don't allow depositing and withdrawing (in this order) in the same block. Thus, if `Vault.rebalance` deposits funds to a TrueFi or Idle vault and then withdraws funds to cover reserved funds, there will always be a revert and rebalancing of the vault will be blocked.



Impact

A rebalancing of a vault can be blocked indefinitely until the vault has enough funds to cover the reserved funds after funds were distributed to protocols.

Code Snippet

1. Vault.rebalance distributes funds to protocols after rebalancing: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L135>
2. executeDeposits deposits funds to protocols: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L147>
3. pullFunds withdraws funds from protocols: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L150>
4. TrueFi vaults revert when depositing and withdrawing in the same block: <https://github.com/truSToken/contracts-pre22/blob/76854d53c5036777286d4392495ef28cd5c5173a/contracts/truefi2/TrueFiPool2.sol#L488>

```
function join(uint256 amount) external override joiningNotPaused {
    ...
    latestJoinBlock[tx.origin] = block.number;
    ...
}

function liquidExit(uint256 amount) external sync {
    require(block.number != latestJoinBlock[tx.origin], "TrueFiPool: Cannot join
↵ and exit in same block");
    ...
}
```

1. Idle vaults revert when depositing and withdrawing in the same block: https://github.com/Idle-Labs/idle-contracts/blob/develop/contracts/mocks/IdleTokenV3_1NoConst.sol#L630

```
function mintIdleToken(uint256 _amount, bool, address _referral)
    external nonReentrant whenNotPaused
    returns (uint256 mintedTokens) {
    _minterBlock = keccak256(abi.encodePacked(tx.origin, block.number));
    ...
}

function _redeemIdleToken(uint256 _amount, bool[] memory _skipGovTokenRedeem)
    internal nonReentrant
    returns (uint256 redeemedTokens) {
    _checkMintRedeemSameTx();
```



```
    ...  
  }  
  
  function _checkMintRedeemSameTx() internal view {  
    require(keccak256(abi.encodePacked(tx.origin, block.number)) != _minterBlock,  
↪    "REE");  
  }
```

Tool used

Manual Review

Recommendation

Consider improving the logic of the `Vault.rebalance` function so that funds are never withdrawn from protocols after they have been deposited to protocols.



Issue M-9: XProvider forces increased relayer fees when transferring tokens cross-chain

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/325>

Found by

Jeiwan

Summary

When transferring tokens cross-chain, XProvider sends two cross-chain message, while only one can be sent. Whoever triggers cross-chain token transfers (which are required to complete rebalancing) will pay relayer fees twice.

Vulnerability Detail

The protocol integrates with Connex to handle cross-chain interactions. XProvider is a contract that manages interactions between vaults deployed on all supported networks and XChainController. XProvider is deployed on each of the network where a vault is deployed and is used to send and receive cross-chain messages via Connex. Among other things, XProvider handles cross-chain token transfers during vaults rebalancing:

1. xTransferToController transfers funds from a vault to the XChainController;
2. xTransferToVaults transfers funds from the XChainController to a vault.

The two functions, besides sending tokens, also update the state in the destination contract:

1. xTransferToController calls XChainController.upFundsReceived to update the counter of vaults that have sent tokens to XChainController;
2. xTransferToVaults calls Vault.receiveFunds to set a "funds received" flag in the vault.

Both sending tokens and triggering a state change send a cross-chain message by calling `IConnex.xcall`:

1. xSend calls IConnex.xcall and sends relayer fee along the call;
2. xTransfer sends tokens by calling IConnex.xcall, and it also requires paying relayer fee.

Thus, the caller of xTransferToController and xTransferToVaults will have to pay double relayer fee. Since these functions are mandatory for rebalancing, the extra fee will have to be paid by the guardian or any actor who manages vaults



rebalancing. However, Connex allows to transfer tokens and make arbitrary calls in one message, while paying relayer fee only once.

Impact

`xTransferToController` and `xTransferToVaults` incur double relayer fees on the caller. The extra cost will have to be paid by whoever manages rebalancing.

Code Snippet

1. `xTransferToController` calls `xTransfer` and `pushFeedbackToXController`—both of them create a cross-chain message: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L321>
2. `xTransferToVaults` calls `xTransfer` and `pushFeedbackToVault`—both of them create a cross-chain message: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L362>

Tool used

Manual Review

Recommendation

According to [the documentation of Connex](#), the `_callData` argument of `xcall` doesn't need to be empty when transferring tokens: `xReceive` can handle both of them together:

```
function xReceive(
    bytes32 _transferId,
    uint256 _amount,
    address _asset,
    address _originSender,
    uint32 _origin,
    bytes memory _callData
) external returns (bytes memory) {
    // Check for the right token
    require(
        _asset == address(token),
        "Wrong asset received"
    );

    // Enforce a cost to update the greeting
    require(
        _amount > 0,
        "Must pay at least 1 wei"
    );
```



```
// Unpack the _callData
string memory newGreeting = abi.decode(_callData, (string));
_updateGreeting(newGreeting);
}
```

Also, judging by the implementation of the Connex contract, the passed calldata is executed even when tokens are transferred:

```
// transfer funds to recipient
AssetLogic.handleOutgoingAsset(_asset, _args.params.to, _amountOut);

// execute the calldata
_executeCalldata(_transferId, _amountOut, _asset, _reconciled, _args.params);
```

Thus, in `xTransferToController` and `xTransferToVaults`, consider passing the calldata of the second calls to `xTransfer`.

Discussion

sjoerdsommen

No funds are at risk here. We don't know if it's smart to fix this. There is a good argument to be made to keep xchain actions as simple as possible, therefore separating the transfer of funds and the messaging.



Issue M-10: Missing transaction expiration check result in reward tokens selling at a lower price

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/323>

Found by

Bauer, Jaiwan, PRAISE

Summary

Selling of reward tokens misses the transaction expiration check, which may lead to reward tokens being sold at a price that's lower than the market price at the moment of a swap. Gamers and depositors may receive less yield than expected.

Vulnerability Detail

The protocol integrates with third-party vaults (e.g. Aave, Compound, Yearn, etc.), in which it deposits user funds to generate yield. Some of the third-party vaults (at the moment, only Compound) reward their users with protocol tokens (COMP, in the case of Compound). These tokens can be claimed and sold for the underlying token of the vaults (e.g. USDC):

1. during rebalancing;
2. at any moment by calling claimTokens.

The Swap.swapTokensMulti function, which is responsible for selling tokens on Uniswap, sets the deadline argument of the `exactInput` call to `block.timestamp`—this basically disables the transaction expiration check because the deadline will be set to whatever timestamp the block including the transaction is mined at.

Transaction expiration check (implemented in Uniswap via the deadline argument) allows users of Uniswap to protect from selling tokens at an outdated price that's lower than the current price. Consider this scenario:

1. The Vault.rebalance function is called on the Ethereum mainnet.
2. Before the transaction is mined, there's a rapid increase of gas cost. The transaction remains in the mempool for some time since the gas cost paid by the transaction is lower than the current gas price.
3. While the transaction is in the mempool, the price of the reward token increases.
4. After a while, gas cost drops and the transaction is mined. Since the value of `amountOutMinimum` was calculated based on an outdated reward token price which is now lower than the current price, the swapping is sandwiched by a



MEV bot. The bot decreases the price of the reward token in a Uniswap pool so that the minimum output amount check still holds and earns a profit from the swapping happening at a lower price.

5. As a result of the sandwich attack, reward tokens are swapped at an outdated price, which is now lower than the current price of the tokens. The protocol (and thus gamers and depositors) earn less yield than they could've received by selling the tokens at the current price.

Impact

Claiming and selling reward tokens can be exploited by a sandwich attack. Gamers and depositors may receive less yield than expected due to reward tokens have been sold at an outdated price.

Code Snippet

1. ClaimTokens sells the reward token for the underlying token:
<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L404>
2. ClaimToken is called during rebalancing: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L141>
3. The proceedings from selling reward tokens increase the total number of underlying tokens during rebalancing: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L158-L164>
4. When Swap.swapTokensMulti calls Uniswap's exactInput, it sets deadline to block.timestamp, which disables the transaction expiration protection:
<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/libraries/Swap.sol#L88>

Tool used

Manual Review

Recommendation

Consider a reasonable value to the deadline argument. For example, Uniswap sets it to 30 minutes on the Ethereum mainnet and to 5 minutes on L2 networks. Also consider letting the guardian and/or the DAO change the value when on-chain conditions change and may require a different value.

Issue M-11: The guardian may not be able to blacklist a protocol

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/320>

Found by

Jeiwan

Summary

An attempt to blacklist a protocol may revert in some situations, not making it possible for the guardian to remove a malicious or a hacked protocol from the system. The protocol will keep receiving funds during rebalancing.

Vulnerability Detail

The `Vault.blacklistProtocol` function allows the guardian to blacklist a protocol: a blacklisted protocol will be excluded from receiving funds during rebalancing. When a protocol is blacklisted, vault funds need to be removed from it and updated the cached amount of funds in the protocol. However, the latter can cause a revert when `balanceUnderlying(_protocolNum)` is greater than `savedTotalUnderlying`, and this can realistically happen in any protocol.

`savedTotalUnderlying` is set at a final step of rebalancing—in the `Vault.rebalance` function: its value is the sum of all funds deposited to underlying protocols. When a protocol is blacklisted, its underlying balance is re-calculated and subtracted from `savedTotalUnderlying`. In situations when the blacklisted protocol is the only protocol that has allocations (allocations are chosen by gamers based on the performance of different protocols, thus one, the most profitable, protocol can be the only protocol of a vault), its current underlying balance will be greater than the cached one (`savedTotalUnderlying`) because the protocol will accrue interest between the moment `savedTotalUnderlying` was calculated and the moment it's being blacklisted. Thus, the `balanceProtocol` variable in the `blacklistProtocol` function can be greater than `savedTotalUnderlying`, which will cause a revert. The function will keep reverting until there's another protocol with allocations or until there are no allocations in the vault at all (which solely depends on gamers, not the guardian).

Impact

The guardian may not be able to blacklist a malicious or a hacked protocol, and the protocol will keep receiving funds during rebalancing.



Code Snippet

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L477-L483>

Tool used

Manual Review

Recommendation

In the `Vault.blacklistProtocol` function, consider checking if `savedTotalUnderlying` is greater or equal to `balanceProtocol`: if it's not so, consider setting its value to 0, instead of subtracting `balanceProtocol` from it.



Issue M-12: Deposited funds are locked in inactive vaults

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/318>

Found by

saidam017, Met, Jeiwan

Summary

Users cannot request withdrawal of their funds from inactive vaults: if a vault receives 0 allocations from gamers, only previous withdrawal requests can be process and new requests cannot be made.

Vulnerability Detail

During rebalancing, if a vault receives 0 allocations from gamers, it becomes inactive. `MainVault` doesn't allow calling these functions on an inactive vault:

1. `deposit`, which makes sense because the vault doesn't generate yield and users shouldn't be allowed to deposit funds in such vaults;
2. `withdraw`, which also makes sense because the vault holds no funds, so immediate withdrawals are not possible;
3. `withdrawalRequest`, which seems wrong because this doesn't allow users to withdraw funds after next rebalancing.

Since inactive vaults don't receive funds during rebalancing, it expected that immediate withdrawals via the `withdraw` function are disabled (inactive vaults don't hold funds, thus funds cannot be withdrawn immediately). However, disallowing `withdrawalRequest` seems like a mistake: users who have deposited their funds into an inactive vault and who are still holding shares of the vault might decide, after seeing that the vault haven't received allocations during last rebalancing (e.g. protocols on other networks might generate higher yields and gamers might want to direct all allocations to those protocols), to withdraw their funds. `withdrawalRequest` allows them to request withdrawal after next rebalancing, which is a crucial feature of the protocol.

Impact

In the worst case scenario, user funds can be locked in an inactive vault for a prolonged period of time. E.g. if the vault is deployed in a network where yield generating protocols produce lower APY than protocols on other networks. In this scenario, gamers will be willing to allocate to more profitable protocols, but depositors of the inactive vault will be forced to keep their funds locked in the vault



until on-chain conditions change (which are not controlled by users, e.g. users cannot force a protocol to have a higher APY).

Code Snippet

1. When `XChainController` receives allocations, it disables vault that have 0 allocations and that haven't received new allocations: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L229-L235>
2. Withdrawals cannot be requested in inactive vaults due to the `onlyWhenVaultIsOn` modifier: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L149-L151>
3. `withdrawAllowance` can only withdraw funds when there's a withdrawal request: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L166-L169>

Tool used

Manual Review

Recommendation

Consider allowing users call `withdrawalRequest` to request withdrawal in inactive vault. During rebalancing, inactive vaults with positive `withdrawalRequest` should receive enough funds for all requests to be processed.

Discussion

sjoerdsommen

Duplicate with <https://github.com/sherlock-audit/2023-01-derby-judging/issues/318> <https://github.com/sherlock-audit/2023-01-derby-judging/issues/211>
<https://github.com/sherlock-audit/2023-01-derby-judging/issues/137>

hrishibhat

Given the necessary preconditions for the issue to happen considering this issue as a valid medium



Issue M-13: Delayed Compound and Beta interest accrual reduces gamer rewards and affects funds distribution to vaults

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/317>

Found by

psy4n0n, hyh, KingNFT, Jeiwan

Summary

During rebalancing, Compound and Beta interest is not accrued until after gamer rewards are calculated leading to the rewards being calculated on the underlying amounts that are smaller than the real amounts. As a result, gamers receive reduced rewards. Also, since interest accrual doesn't happen when vaults push underlying amounts to `XChainController`, the underlying amounts to be distributed among vaults will be smaller than real amounts.

Vulnerability Detail

The Derby protocol allows users to deposit funds to third-party protocols to earn passive yield. Two of such protocols are Compound and Beta: funds deposited to Compound and Beta earn interest collected from borrowers who borrow the funds. Due to accruing of interest, the exchange rate of cTokens and BTokens increases over time: for example, 1 cToken can be exchanged for 1.01 underlying token after some time.

During rebalancing, Derby vaults report their underlying token balances (the amount of funds deposited in third-party protocols) to `XChainController`, which re-allocates and re-distributes them according to the allocations set by gamers. However, when the underlying balances are calculated, interest is not accrued on Compound and Beta:

1. `setTotalUnderlying` calls `balanceUnderlying` on each of the supported protocol;
2. `CompoundProvider.balanceUnderlying` multiplies contract's cToken balance by the exchange rate of the cToken;
3. `CompoundProvider.exchangeRate` calls `ICToken.exchangeRateStored`, which doesn't accrue interest and returns cached exchange rate:

This function does not accrue interest before calculating the exchange rate



1. Likewise, `BetaProvider.balanceUnderlying` and `BetaProvider.calcShares` read `IBeta.totalLoanable` and `IBeta.totalLoan` to convert BTokens to underlying tokens, however the interest is not accrued in `totalLoan` beforehand (as can be seen in the code of `BToken`, [accruing interest increases totalLoan](#)).

Thus, the interest accrued by the funds deposited to Compound and Beta since the previous rebalancing won't be counted in the new rebalancing, and the underlying balance reported by the vault will be lower than the real balance.

Interest is accrued only at the end of rebalancing, in the `Vault.rebalance` function: first `deposit` or `withdrawal` to/from Compound and Beta will accrue interest. However, this will happen after gamer rewards have been calculated:

1. gamer rewards are calculated in `storePriceAndRewards`, before funds are deposited/withdrawn to Compound or Beta;
2. gamer rewards are calculated based on the underlying balance calculated in `calcUnderlyingIncBalance`, which reads the value of `savedTotalUnderlying`—it was set in the beginning of rebalancing, and interest wasn't accrued before it was set.

Thus, gamer rewards will always lag behind actual underlying balances of Compound and Beta, and gamers will always earn reduced rewards.

Impact

Gamer receive reduced rewards due to delayed accruing of interest in Compound and Beta.

Code Snippet

1. `setTotalUnderlying` is called when vaults report their balances: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L252>
2. `setTotalUnderlying` calls `balanceUnderlying`, which calls `balanceUnderlying` on each protocol provider: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L344-L361>
3. `CompoundProvider.balanceUnderlying` calls `exchangeRate` to get the exchange rate of the `cToken`: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/CompoundProvider.sol#L96>
4. The exchange rate is read via the `exchangeRateStored` function, which returned cached rate: the interest earned since the previous accrual is not counted: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/CompoundProvider.sol#L127>
<https://github.com/compound-finance/compound-protocol/blob/master/contracts/CToken.sol#L284-L293>



5. `BetaProvider.balanceUnderlying` and `BetaProvider.calcShares` read `IBeta.totalLoan` to convert BTokens to underlying tokens:
<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimizer/contracts/Providers/BetaProvider.sol#L89>
<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimizer/contracts/Providers/BetaProvider.sol#L102>
6. In `BToken`, `totalLoan` is increased by accumulated interest when interest is accrued: <https://github.com/beta-finance/beta/blob/master/contracts/BToken.sol#L92-L93>

Tool used

Manual Review

Recommendation

In `CompoundProvider.exchangeRate`, consider calling `exchangeRateCurrent` instead of `exchangeRateStored`. In `BetaProvider.balanceUnderlying` and `BetaProvider.calcShares` consider calling `IBeta accrue` before calling `IBeta.totalLoan`.

Discussion

sjoerdsommen

This does only lead to a slightly lower yield. If we would fix it, this would also lead to a more expensive call. We would have to investigate this.

hrishibhat

Considering this issue a valid medium as there is a stale `exchangeRateStored` resulting in reduced rewards for depositors and affects allocator rewards.



Issue M-14: Gamers will not receive rewards for allocating to Aave and Beta vaults, while the vaults do generate yield

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/313>

Found by

psy4n0n, c7e7eff, Ch_301, hyh, foxb868, cergyk, Jeiwan

Summary

Since `AaveProvider` and `BetaProvider` don't track the exchange rate of respective protocol tokens, gamer rewards will always be 0 for these protocols. The protocols, however, will generate yield that will be distributed only to depositors.

Vulnerability Detail

Gamers are participants who allocate funds to yield generating protocols; as a reward, they receive a share of the yield. Depositing and withdrawing of funds, as well as accruing of rewards, happens during rebalancing of vaults. Vaults can be rebalanced once in two weeks, and, between rebalancings, funds generate yield in third-party protocols.

The yield generated by protocols is tracked during rebalancings using the exchange rate of a protocol: since protocol generate yield, the exchange rate always increases, and the difference in exchange rates between two rebalancings is used to calculate the amount of yield generated between the rebalancings.

However, `AaveProvider.exchangeRate` and `BetaProvider.exchangeRate` return wrong exchange rates:

1. `AaveProvider` always returns 1;
2. `BetaProvider` always returns 0.

If the exchange rate is always 1, then the numerator of the gamer rewards calculation formula will always be 0, and gamer rewards will always be 0. If the exchange rate is 0, gamer rewards won't be calculated at all.

Impact

Gamers who allocate tokens to Aave and Beta vaults (while these vaults may generate high yield, compared to other protocols) will not receive rewards. The entire yield generated by the vaults will be distributed only among vault depositors.



Code Snippet

1. The change in protocol token price determines gamer rewards:
<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimizer/contracts/Vault.sol#L233-L236>
2. AaveProvider always return the exchange rate of 1:
<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimizer/contracts/Providers/AaveProvider.sol#L112>
3. BetaProvider always returns the exchange rate of 0:
<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimizer/contracts/Providers/BetaProvider.sol#L118>

Tool used

Manual Review

Recommendation

Consider correctly tracking the exchange rate of Aave and Beta tokens in the respective providers.



Issue M-15: Gamer rewards are reduced due to fund pulls before accrual of rewards

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/312>

Found by

Auditwolf, Jeiwan

Summary

During rebalancing, funds can be pulled from yield generating protocols before rewards (reward per DERBY token) were calculated. Gamers may receive reduced rewards even though the allocated funds remained in protocols over an entire period.

Vulnerability Detail

Gamers are participants who allocate funds to yield generating protocols; as a reward, they receive a share of the yield. Depositing and withdrawing of funds, as well as accruing of rewards, happens during rebalancing of vaults. Vaults can be rebalanced once in two weeks, and, between rebalancings, funds generate yield in third-party protocols.

To track how much yield was generated, and how much rewards gamers will receive, it's require to know the amount of funds deposited into a vault. When rebalancing of a vault starts, the vault sends its balances (including the funds deposited to third-party protocols) to `XChainController` via the `MainVault.pushTotalUnderlyingToController` function. The function calls `setTotalUnderlying`, which computes the funds deposited to protocols and caches the amount in the `savedTotalUnderlying` variable. Following the logic of yield generation and the accrual of gamer rewards, the share of yield their will receive should be generated from `savedTotalUnderlying` (it's the amount of funds that have been generating yield in protocols since the previous rebalancing). However, `savedTotalUnderlying` can be reduced before gamer rewards are calculated, which reduces the rewards.

After a vault has received re-allocated fund amounts from `XChainController`, it the `MainVault.rebalanceXChain` function must be called: the function will calculated and send the token amounts that the vault must remove from protocols and send to other vaults via `XChainController`. If there's not enough funds in the vault, it'll pull funds from protocols and reduce `savedTotalUnderlying`. Later, when gamer rewards are calculated at a later rebalancing step, `savedTotalUnderlying` will be smaller than it was when rebalancing started and gamer rewards will be calculated on that smaller `savedTotalUnderlying`.



In the current implementation, gamer rewards are calculated on the updated vault and protocol balances. The `Vault.rebalance` function is called at the end of a rebalancing of a vault, after the vault has already sent and received tokens from `XChainController`. This is wrong because these amount haven't yet generated yield:

1. if, as a result of a rebalancing, the vault has received more funds, the calculated game rewards will be bigger since the protocols haven't yet generated yield and yield may be smaller than expected (APY is calculated based on past results);
2. if the vault has received less funds, the calculated rewards will be smaller since the rewards are calculated on a reduced amount.

Impact

Gamers may receive reduced rewards when funds are pulled from yield-generating protocols after a rebalancing has started and before rewards were calculated. Depending on the amount pulled from protocols (which depends on current and new allocations, the cross-chain accounting in `XChainController`, and balances of other vaults) earning of gamers may be significantly reduced. In the worst case scenario, gamers may even be forced to pay the negative fee penalty when removing allocations.

Code Snippet

1. `savedTotalUnderlying` is calculated and set in the beginning of a rebalancing: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L252>
2. Before gamer rewards have been calculated, `savedTotalUnderlying` can be reduced: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L310>
3. When calculating gamer rewards, a reduced value of `savedTotalUnderlying` may be used, instead of the value at the beginning of the rebalancing: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L144>
4. Gamer rewards calculation requires the amount of funds that were used to generate yield: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L234>

Tool used

Manual Review



Recommendation

Consider calculating gamer rewards in the `MainVault.pushTotalUnderlyingToController` function, after calling `setTotalUnderlying`. This way, gamer rewards will be calculated on the exact amount of funds deposited into yield-generating protocols based on the allocations suggested by gamers.

This also fixes another problem: in the current implementation, protocol token price change is applied to the updated (rebalanced) vault balance, however the yield is generated on the previous balance (before it's rebalanced). When calculating gamer rewards, protocol token price change is multiplied by the updated vault balance but it should be multiplied by the balance the vault (specifically, a protocol the vault deposited funds to) had at the beginning of the rebalancing.

Discussion

sjoerdsommen

No user funds are at risk, only the rewards will be somewhat lower than the percentage we set beforehand.

sjoerdsommen

Duplicate with #281



Issue M-16: Rebalancing can be indefinitely blocked due to ever-increasing `totalWithdrawalRequests`, causing locking of funds in vaults

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/311>

Found by

nobody2018, SPYBOY, Jeiwan, evan

Summary

Rebalancing can get stuck indefinitely at the `pushVaultAmounts` step due to an error in the accounting of `totalWithdrawalRequests`. As a result, funds will be locked in vaults since requested withdrawals are only executed after a next successful rebalance.

Vulnerability Detail

Funds deposited to underlying protocols can only be withdrawn from vaults after a next successful rebalance:

1. a depositor has to make a withdrawal request first, which is tracked in the current rebalance period;
2. requested funds can be withdrawn in the next rebalance period.

Thus, it's critical that rebalancing doesn't get stuck during one of its stages.

During rebalancing, vaults report their balances to `XChainController` via the `pushTotalUnderlyingToController` function: the function sends the current unlocked (i.e. excluding reserved funds) underlying token balance of the vault and the total amount of withdrawn requests in the current period. The latter amount is stored in the `totalWithdrawalRequests` storage variable:

1. the variable is increased when a new withdrawal request is made;
2. and it's set to 0 after the vault has been rebalanced—its value is added to the reserved funds.

The logic of `totalWithdrawalRequests` is that it tracks only the requested withdrawal amounts in the current period—this amount becomes reserved during rebalancing and is added to `reservedFunds` after the vault has been rebalanced.

When `XChainController` receives underlying balances and withdrawal requests from vaults, it tracks them internally. The amounts then used to calculate how much tokens a vault needs to send or receive after a rebalancing: the total withdrawal amount is subtracted from vault's underlying balance so that



it's excluded from the amounts that will be sent to the protocols and so that it could then be added to the reserved funds of the vault.

However, `totalWithdrawalRequests` in `XChainController` is not reset between rebalancings: when a new rebalancing starts, `XChainController` receives allocations from the Game and calls `resetVaultUnderlying`, which resets the underlying balances receive from vaults in the previous rebalancing. `resetVaultUnderlying` doesn't set `totalWithdrawalRequests` to 0:

```
function resetVaultUnderlying(uint256 _vaultNumber) internal {
    vaults[_vaultNumber].totalUnderlying = 0;
    vaultStage[_vaultNumber].underlyingReceived = 0;
    vaults[_vaultNumber].totalSupply = 0;
}
```

This cause the value of `totalWithdrawalRequests` to accumulate over time. At some point, the total historical amount of all withdrawal requests (which `totalWithdrawalRequests` actually tracks) will be greater than the underlying balance of a vault, and this line will revert due to an underflow in the subtraction:

```
uint256 totalUnderlying = getTotalUnderlyingVault(_vaultNumber) -
↳ totalWithdrawalRequests;
```

Impact

Due to accumulation of withdrawal request amounts in the `totalWithdrawalRequests` variable, `XChainController.pushVaultAmounts` can be blocked indefinitely after the value of `totalWithdrawalRequests` has grown bigger than the value of `totalUnderlying` of a vault. Since withdrawals from vaults are delayed and enable in a next rebalancing period, depositors may not be able to withdraw their funds from vaults, due to a block rebalancing.

While `XChainController` implements a bunch of functions restricted to the guardian that allow the guardian to push a rebalancing through, neither of these functions resets the value of `totalWithdrawalRequests`. If `totalWithdrawalRequests` becomes bigger than `totalUnderlying`, the guardian won't be able to fix the state of `XChainController` and push the rebalancing through.

Code Snippet

1. When a rebalancing starts, previously reported underlying balances are reset: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimizer/contracts/XChainController.sol#L212>
2. However, `totalWithdrawalRequests` is never reset: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimizer/contracts/XChainController.sol#L212>



[miser/contracts/XChainController.sol#L180-L182](#)

3. When `XChainController` calculates new amounts of underlying tokens, it subtracts `totalWithdrawalRequests` from `totalUnderlying` to reserve requested amounts in the current period:
[https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L303](#)
4. `totalWithdrawalRequests` is always increased when a vault reports its balances: [https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L287](#)

Tool used

Manual Review

Recommendation

In `XChainController.resetVaultUnderlying`, consider setting `vaults[_vaultNumber].totalWithdrawalRequests` to 0. `totalWithdrawalRequests`, like its `MainVault.totalWithdrawalRequests` counterpart, tracks withdrawal requests only in the current period and should be reset to 0 between rebalancings.

Discussion

sjoerdsommen

`totalWithdrawalRequests` in the `xChainController` is never reset.

sjoerdsommen

duplicate with #239



Issue M-17: Protocol is will not work on most of the supported blockchains due to hardcoded WETH contract address.

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/308>

Found by

gogo

Summary

The WETH address is hardcoded in the Swap library.

Vulnerability Detail

As stated in the README.md, the protocol will be deployed on the following EVM blockchains - Ethereum Mainnet, Arbitrum, Optimism, Polygon, Binance Smart Chain. While the project has integration tests with an ethereum mainnet RPC, they don't catch that on different chains like for example Polygon several functionalities will not actually work because of the hardcoded WETH address in the Swap.sol library:

```
address internal constant WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/libraries/Swap.sol#L28>

Impact

Protocol will not work on most of the supported blockchains.

Code Snippet

```
address internal constant WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/libraries/Swap.sol#L28>

Tool used

Manual Review



Recommendation

The WETH variable should be immutable in the Vault contract instead of a constant in the Swap library and the Wrapped Native Token contract address should be passed in the Vault constructor on each separate deployment.



Issue M-18: Should not blacklist protocol when totalUn- derlying is not withdrawn.

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/280>

Found by

spyrosonic10, Ch_301

Summary

`blacklistProtocol()` method does not validate that all amount is withdrawn from the protocol

Vulnerability Detail

Guardian can call `blacklistProtocol()` method of `Vault.sol`. `balanceProtocol` is total balance in the protocol and this amount is withdrawn from protocol into the vault in `blacklistProtocol()`. It is not guaranteed that each protocol successfully withdraw requested amount. This method will mark protocol blacklisted no matter full amount is withdrawn or not.

Impact

If partial withdraw happen during call to `blacklistProtocol()` then fund will stuck in protocol unless Dao add same protocol again.

Code Snippet

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L477-L483>

Tool used

Manual Review

Recommendation

Consider implementing something along these lines

- `blacklistProtocol` function can be used to
 - call `withdrawFromProtocol`
 - and update allocation to zero



- and check `if(balanceUnderlying() > 0)` then only update `totalUnderlying` as such and update state/flag to user later as needed.
 - and mark blacklisted
- A dedicated withdraw function to withdraw fund from protocol which is
 - blacklisted
 - and has 0 allocation
 - and has non-zero LP tokens in Vault.

Discussion

sjoerdsommen

duplicate with <https://github.com/sherlock-audit/2023-01-derby-judging/issues/280> <https://github.com/sherlock-audit/2023-01-derby-judging/issues/217>



Issue M-19: Vault::pullFunds doesn't pull funds from underlying providers correctly

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/273>

Found by

immeas, spyrosonic10, hyh

Summary

If there isn't enough "free" funds in the vault when needed funds can be pulled from underlying providers. However this might not pull even though there is enough funds.

Vulnerability Detail

pullFunds is intended to pull funds from underlying providers if there either isn't enough liquidity to pay withdrawals or do a rebalance.

It works such that it will pull funds from underlying providers until there is enough funds available in the vault: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L111-L127>

Now, imagine that there are two underlying providers with these allocations [1000,1000e6], minimumPull is 1e6 and the value needed to pull is 100e6 and for simplicity there's no existing funds in the vault so

```
vaultCurrency.balanceOf(address(this)) = 0;
```

pullFunds will check the first provider, since there is an allocation this row passes: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L114>

```
next is uint256 amountToWithdraw = shortage > balanceProtocol ?
balanceProtocol : shortage
```

shortage is 100e6 and balance protocol 1000 so amountToWithdraw will be 1000.

1000 will fail on the check if (amountToWithdraw < minimumPull) break; and leave the loop and pullFunds early even though there were funds available in the second vault.

If the need to pull funds from underlying providers was because of withdrawal request/rewards this might make it impossible for users to withdraw. Since the protocol still believes (rightfully) that there is enough underlying. However it cannot be pulled.



It is also possible for a malicious user to use this to grief as they can allocate very small amounts to "early" providers if they don't have any allocations.

Impact

Users might not be able to withdraw or funds not balanced properly because there is a low allocation to an underlying provider.

Code Snippet

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L122>

Tool used

Manual Review

Recommendation

I recommend these changes:

```
diff --git a/derby-yield-optimiser/contracts/Vault.sol
↪ b/derby-yield-optimiser/contracts/Vault.sol
index ac9020d..9466d2c 100644
--- a/derby-yield-optimiser/contracts/Vault.sol
+++ b/derby-yield-optimiser/contracts/Vault.sol
@@ -114,12 +114,14 @@ contract Vault is ReentrancyGuard {
     if (currentAllocations[i] == 0) continue;

    uint256 shortage = _value - vaultCurrency.balanceOf(address(this));
+   if(shortage < minimumPull) break; // if not enough is missing, leave
+
    uint256 balanceProtocol = balanceUnderlying(i);

    uint256 amountToWithdraw = shortage > balanceProtocol ? balanceProtocol :
↪ shortage;
    savedTotalUnderlying -= amountToWithdraw;

-    if (amountToWithdraw < minimumPull) break;
+    if (amountToWithdraw < minimumPull) continue; // if not enough in this
↪ protocol, check next
    withdrawFromProtocol(i, amountToWithdraw);

    if (_value <= vaultCurrency.balanceOf(address(this))) break;
```



Issue M-20: XChainController::sendFundsToVault can be grieved and leave XChainController in a bad state

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/271>

Found by

nobody2018, immeas, Ch_301, saidam017, cergyk

Summary

A user can grief the send funds to vault state transition during by calling sendFundsToVault multiple times with the same vault.

Vulnerability Detail

During rebalancing, some vaults might need funds sent to them. They will be in state WaitingForFunds. To transition from here any user can trigger XChainController to send them funds by calling sendFundsToVault.

This is trigger per chain and will transfer funds from XChainController to the respective vaults on each chain.

At the end, when the vaults on each chain are processed and either have gotten funds sent to them or didn't need to sendFundsToVaults will trigger the state for this vaultNumber to be reset.

However, when transferring funds, there's never any check that this chain has not already been processed. So any user could simply call this function for a vault that either has no funds to transfer or where there's enough funds in XChainController and trigger the state reset for the vault.

PoC in xChaincontroller.test.ts, run after 4.5) Trigger vaults to transfer funds to xChainController:

```
it('5) Grief xChainController send funds to vaults', async function () {
  await xChainController.sendFundsToVault(vaultNumber, slippage, 10000, 0, {
    value: 0, });
  await xChainController.sendFundsToVault(vaultNumber, slippage, 10000, 0, {
    value: 0, });
  await xChainController.sendFundsToVault(vaultNumber, slippage, 10000, 0, {
    value: 0, });
  await xChainController.sendFundsToVault(vaultNumber, slippage, 10000, 0, {
    value: 0, });

  expect(await
    xChainController.getFundsReceivedState(vaultNumber)).to.be.equal(0);
```



```

    expect(await vault3.state()).to.be.equal(3);

    // can't trigger state change anymore
    await expect(xChainController.sendFundsToVault(vaultNumber, slippage, 1000,
↪   relayerFee, {value: parseEther('0.1'),})).to.be.revertedWith('Not all funds
↪   received');
  });

```

Impact

XChainController ends up out of sync with the vault(s) that were supposed to receive funds.

guardian can resolve this by resetting the states using admin functions but these functions can still be frontrun by a malicious user.

Until this is resolved the rebalancing of the impacted vaults cannot continue.

Code Snippet

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L439>

Tool used

Manual Review, hardhat

Recommendation

I recommend the protocol either keeps track of which vaults have been sent funds in XChainController.

or changes so a vault can only receive funds when waiting for them:

```

diff --git a/derby-yield-optimiser/contracts/MainVault.sol
↪   b/derby-yield-optimiser/contracts/MainVault.sol
index 8739e24..d475ee6 100644
--- a/derby-yield-optimiser/contracts/MainVault.sol
+++ b/derby-yield-optimiser/contracts/MainVault.sol
@@ -328,7 +328,7 @@ contract MainVault is Vault, VaultToken {
    /// @notice Step 5 end; Push funds from xChainController to vaults
    /// @notice Receiving feedback from xController when funds are received, so
↪   the vault can rebalance
    function receiveFunds() external onlyXProvider {
-     if (state != State.WaitingForFunds) return;
+     require(state == State.WaitingForFunds, stateError);

```



```
    settleReservedFunds();  
}
```

Discussion

sjoerdsommen

Duplicate with #163



Issue M-21: vault stakers and game players share the same reward pool

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/269>

Found by

immeas, rvierdiiev

Summary

Both vault stakers earning yield and game players are taking rewards from the same reward pool. One will take rewards from the other when withdrawing/redeeming.

Vulnerability Detail

Lets simplify the whole setup a bit to show the issue here. Imagine one vault on one chain with one provider. Alice plays the game and deposits 10 DRB all allocated to the provider. Bob deposits 10k USDC to the vault.

Time passes the provider has a 10% increase. The funds are now 11k.

Both Bob and Alice are happy with this and want to collect their rewards. Alice redeems her rewards and Bob registers for a withdraw (or just withdraws).

The issue here is that both have claim to the same 1k rewards. Bob who staked should get them as yield for staking and Alice should get them as rewards for playing.

If both manage to withdraw/redeem in the same cycle this is where it will break down: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L303>

As the `totalWithdrawalRequests` (Bobs withdraw of all his shares and Alice redeeming of rewards) will be larger than `getTotalUnderlyingVault`.

The issue is not with this line of code though, its with the design of the rewards as game players and vault stakers both have a claim on the same rewards.

Impact

If a staker in the vault withdraws they will take rewards from game players and if a game player they will take yield from someone staking. If both at the same time, they will take rewards from user still staking. Resulting in the "last" user possibly not being able to withdraw due to not enough funds.



Code Snippet

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XChainController.sol#L303>

Tool used

Manual Review, hardhat

Recommendation

Honestly I don't see an easy way out of this: You could split the pools, but why would anyone stake in the vault then? Since that is just staking with less yield compared to staking in the underlying protocols directly.

You could limit staking to game players, and then make them only split the funds they actually stake. That's a large change to the protocol though and would remove the use for Derby tokens.

You could also limit rewards to only Derby tokens, but that only works as long as there is DRB to hand out. It would also require some change to the way the rewards are reserved as currently you can create a reserved rewards greater than underlying thus underflowing in stage 3.

Discussion

sjoerdsommen

Implicitly the rewards that have been paid out are accounted for in the exchangerate. However, there are some situations, like in this example, where this does not work anymore. Fixing it will be very gas intensive so we would have to see how to do it.

sjoerdsommen

Duplicate with #143



Issue M-22: Vault could `rebalance()` before funds arrive from `xChainController`

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/268>

Found by

Ch_301

Summary

Invoke `sendFundsToVault()` to Push funds from `xChainController` to vaults. which is call `xTransferToVaults()`

For the cross-chain rebalancing `xTransferToVaults()` will execute this logic

```
...
pushFeedbackToVault(_chainId, _vault, _relayerFee);
xTransfer(_asset, _amount, _vault, _chainId, _slippage, _relayerFee);
...
```

- `pushFeedbackToVault()` Is to invoke `receiveFunds()` `pushFeedbackToVault()` always travel through the slow path
- `xTransfer()` to transfer funds from one chain to another If fast liquidity is not available, the `xTransfer()` will go through the **slow path**.

The vulnerability is if the `xcall()` of `pushFeedbackToVault()` excited successfully before `xTransfer()` transfer the funds to the vault, anyone can invoke `rebalance()` this will lead to rebalancing Vaults with Imperfect funds (this could be true only if funds that are expected to be received from `XChainController` are greater than `reservedFunds` and `liquidityPerc` together)

Vulnerability Detail

The above scenario could be done in two possible cases 1- `xTransfer()` will go through the **slow path** but because High Slippage the cross-chain message will wait until slippage conditions improve (relayers will continuously re-attempt the transfer execution).

2- Connex Team says

```
All messages are added to a Merkle root which is sent across chains every 30 mins
And then those messages are executed by off-chain actors called routers
```

```
so it is indeed possible that messages are received out of order (and
↳ potentially with increased latency in between due to batch times)
```



```
For "fast path" (unauthenticated) messages, latency is not a concern, but
↳ ordering may still be (this is an artifact of the chain itself too btw)
one thing you can do is add a nonce to your messages so that you can yourself
↳ order them at destination
```

so `pushFeedbackToVault()` and `xTransfer()` could be added to a different Merkle root and this will lead to executing `receiveFunds()` before funds arrive.

Impact

The vault could `rebalance()` before funds arrive from `xChainController`, this will reduce rewards

Code Snippet

Tool used

Manual Review

Recommendation

Check if funds are arrived or not



Issue M-23: Wrong calculation of `balanceBefore` and `balanceAfter` in deposit method

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/262>

Found by

Ruhum, chaduke, spyrosonic10

Summary

Deposit method calculate net amount transferred from user. It use `reservedFunds` also in consideration when calculating `balanceBefore` and `balanceAfter` but it is not actually require.

```
uint256 balanceBefore = getVaultBalance() - reservedFunds;
vaultCurrency.safeTransferFrom(msg.sender, address(this), _amount);
uint256 balanceAfter = getVaultBalance() - reservedFunds;
uint256 amount = balanceAfter - balanceBefore;
```

Vulnerability Detail

Deposit may fail when `reservedFunds` is greater than `getVaultBalance()`

Impact

Deposit may fail when `reservedFunds` is greater than `getVaultBalance()`

Code Snippet

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L116-L118>

```
uint256 balanceBefore = getVaultBalance() - reservedFunds;
vaultCurrency.safeTransferFrom(msg.sender, address(this), _amount);
uint256 balanceAfter = getVaultBalance() - reservedFunds;
uint256 amount = balanceAfter - balanceBefore;
```

Tool used

Manual Review



Recommendation

Use below code. This is correct way of finding net amount transfer by depositor

```
uint256 balanceBefore = getVaultBalance();  
vaultCurrency.safeTransferFrom(msg.sender, address(this), _amount);  
uint256 balanceAfter = getVaultBalance();  
uint256 amount = balanceAfter - balanceBefore;
```

Discussion

sjoerdsommen

duplicate with #262 #66 #45



Issue M-24: Players can call rebalanceBasket before rewards have been pushed to the game

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/259>

Found by

oot2k, nobody2018, rvierdiev, evan, Jeiwan

Summary

It's possible to manipulate game.sol to a state where players can call rebalanceBasket before rewards have been settled. In some cases, it's possible for a malicious user to profit. In other cases, it's possible for a malicious user to create a period where players lose a significant amount of rewards when they call rebalanceBasket.

Vulnerability Detail

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L343> Observe that vault's receiveProtocolAllocations can be called by the xProvider regardless what state the vault is in.

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L407> <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L472> This means that game.sol's pushAllocationsToVaults can be called immediately after pushAllocationsToController.

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L476> PushAllocationsToVaults resets isXChainRebalancing.

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L324> So if someone call PushAllocationsToVaults for vaults on all the chains, players can rebalanceBasket again.

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L442> Vault's rebalancingPeriod on game.sol is incremented in pushAllocationsToController, which is at the very beginning of the rebalancing process.

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L362> The reward for this rebalancingPeriod is not pushed to the game until the very end of the rebalancing process.

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L390> The problem with calling rebalanceBasket before the



reward for the current rebalancingPeriod is pushed, is that the value of the currentReward is 0 (hasn't been set yet) so the reward calculation is wrong. Depending on the actual value of currentReward, the user can make a profit or incur a loss.

Impact

Consider the following scenario. A vault is ready for rebalancing, so the malicious user calls pushAllocationsToController, and then calls pushAllocationsToVaults for vaults on all chains immediately after.

The period from now to when the rebalancing process finishes is a time frame where reward calculation is wrong (currentReward is 0) but rebalanceBasket can still be called. A unaware user can call rebalanceBasket and get a completely different reward than they are supposed to. More often than not, this is unfavorable since currentReward should usually be positive. The length of this period depends on how fast the rebalancing process, which can be delayed by a variety of factors, completes. As discussed in my other reports, there are various ways to interrupt the rebalancing process.

But regardless of how short this period is, the malicious user can predetermine the actual value of currentReward. If it's negative, then they immediately call rebalanceBasket and get a higher reward than they are supposed to.

Code Snippet

See Vulnerability Detail.

Tool used

Manual Review

Recommendation

Prevent rebalanceBasket from being called before rewards for the current period have been settled.

Issue M-25: inflate initial share price by initial depositor

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/258>

Found by

chainNue, immeas, SPYBOY

Summary

initial deposit can be front-run by non-whitelist address to inflate share price evading the `training` block, then all users after the first (the attacker) will receive no shares in return for their deposit.

Vulnerability Detail

`training` block inside `deposit` function intended to be set as `true` right after deployment. This `training` variable is to make sure the early depositor address is in the whitelist, thus negating any malicious behaviour (especially the first initial depositor)

```
File: MainVault.sol
106:   function deposit(
107:       uint256 _amount,
108:       address _receiver
109:   ) external nonReentrant onlyWhenVaultIsOn returns (uint256 shares) {
110:       if (training) {
111:           require(whitelist[msg.sender]);
112:           uint256 balanceSender = (balanceOf(msg.sender) * exchangeRate) / (10
113:   ↪   ** decimals());
113:           require(_amount + balanceSender <= maxTrainingDeposit);
114:       }
```

First initial depositor issue is pretty well-known issue in vault share-based token minting for initial deposit which is susceptible to manipulation. This issue arise when the initial vault balance is 0, and initial depositor (attacker) can manipulate this share accounting by donating small amount, thus inflate the share price of his deposit. There are a lot of findings about this initial depositor share issue.

Even though the `training` block is (probably) written to mitigate this initial deposit, but since the execution of setting the `training` to be `true` is not in one transaction, then it's possible to be front-run by attacker. Then this is again, will make the initial deposit susceptible to attack.

The attack vector and impact is the same as [TOB-YEARN-003](#), where users may not receive shares in exchange for their deposits if the total asset amount has been



manipulated through a large “donation”.

The initial exchangeRate is a fixed value set on constructor which is not related to totalSupply, but later it will use this totalSupply

```
File: MainVault.sol
64:     exchangeRate = _uScale;
...
290:     function setXChainAllocationInt(
291:         uint256 _amountToSend,
292:         uint256 _exchangeRate,
293:         bool _receivingFunds
294:     ) internal {
295:         amountToSendXChain = _amountToSend;
296:         exchangeRate = _exchangeRate;
297:
298:         if (_amountToSend == 0 && !_receivingFunds) settleReservedFunds();
299:         else if (_amountToSend == 0 && _receivingFunds) state =
↳ State.WaitingForFunds;
300:         else state = State.SendingFundsXChain;
301:     }

File: XChainController.sol
303:     uint256 totalUnderlying = getTotalUnderlyingVault(_vaultNumber) -
↳ totalWithdrawalRequests;
304:     uint256 totalSupply = getTotalSupply(_vaultNumber);
305:
306:     uint256 decimals = xProvider.getDecimals(vault);
307:     uint256 newExchangeRate = (totalUnderlying * (10 ** decimals)) /
↳ totalSupply;
```

Impact

initial depositor can inflate share price, other user (next depositor) can lost their asset

Code Snippet

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L110-L114>

```
File: MainVault.sol
106:     function deposit(
107:         uint256 _amount,
108:         address _receiver
109:     ) external nonReentrant onlyWhenVaultIsOn returns (uint256 shares) {
110:         if (training) {
```



```

111:         require(whitelist[msg.sender]);
112:         uint256 balanceSender = (balanceOf(msg.sender) * exchangeRate) / (10
↳ ** decimals());
113:         require(_amount + balanceSender <= maxTrainingDeposit);
114:     }
115:
116:     uint256 balanceBefore = getVaultBalance() - reservedFunds;
117:     vaultCurrency.safeTransferFrom(msg.sender, address(this), _amount);
118:     uint256 balanceAfter = getVaultBalance() - reservedFunds;
119:
120:     uint256 amount = balanceAfter - balanceBefore;
121:     shares = (amount * (10 ** decimals())) / exchangeRate;
122:
123:     _mint(_receiver, shares);
124: }

```

Tool used

Manual Review

Recommendation

The simplest way around for this is just set the initial `training` to be `true` either in the variable definition or set it in constructor, so the initial depositor will be from the whitelist.

or, more common solution for this issue is, require a minimum size for the first deposit and burn a portion of the initial shares (or transfer it to a secure address)

Discussion

sjoerdsommen

Duplicate with #274



Issue M-26: Vault can lose rewards due to lack of precision

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/238>

Found by

hyh

Summary

Vault's `storePriceAndRewards()` can accrue no rewards as precision can be lost when `price()` has low decimals.

For example, `price()` has 6 decimals for Idle USDC and USDT strategies.

Vulnerability Detail

Suppose `storePriceAndRewards()` is called for Idle USDC protocol, this USDC Vault is relatively new and `Totalunderlying = TotalUnderlyingInProtocols - BalanceVault = USD 30k`, performance fee is 5% and it's 1 mln Derby tokens staked, i.e. `totalAllocatedTokens = 1_000_000 * 1e18`, while `lastPrices[_protocolId] = 1_100_000` (which is 1.1 USDC, Idle has price scaled by underlying decimals and tracks accumulated strategy share price).

Let's say this provider shows stable returns with 1.7% APY, suppose market rates are low and this is somewhat above market. In bi-weekly terms it can correspond to `priceDiff = 730`, as, roughly excluding price appreciation that doesn't influence much here, $(730.0 / 1100000 + 1)^{26} - 1 = 1.7\%$.

In this case `rewardPerLockedToken` will be $30000 * 10^{18} * 5\% * 730 / (1_000_000 * 1_100_000 * 100) = 0$ for all rebalancing periods, i.e. no rewards at all will be allocated for the protocol despite it having positive performance.

Impact

Rewards can be lost for traders who allocated to protocols where `price()` has low decimals.

Code Snippet

Vault's `storePriceAndRewards()` calculates strategy performance reward as `nominator / denominator`, where `nominator = _totalUnderlying * performanceFee * priceDiff`:



<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L220-L245>

```
/// @notice Stores the historical price and the reward per rounded locked token,
↳ ignoring decimals.
/// @dev formula yield protocol i at time t:  $y(it) = (P(it) - P(it-1)) / P(it-1)$ .
/// @dev formula rewardPerLockedToken for protocol i at time t:  $r(it) = y(it) * TVL(t) * perfFee(t) / totalLockedTokens(t)$ 
↳ multiply this ( $r(it)$ ) by the locked tokens on protocol i at time t
/// @param _totalUnderlying Totalunderlying = TotalUnderlyingInProtocols - BalanceVault.
/// @param _protocolId Protocol id number.
function storePriceAndRewards(uint256 _totalUnderlying, uint256 _protocolId)
↳ internal {
    uint256 currentPrice = price(_protocolId);
    if (lastPrices[_protocolId] == 0) {
        lastPrices[_protocolId] = currentPrice;
        return;
    }

    int256 priceDiff = int256(currentPrice - lastPrices[_protocolId]);
    int256 nominator = (int256(_totalUnderlying * performanceFee) * priceDiff);
    int256 totalAllocatedTokensRounded = totalAllocatedTokens / 1E18;
    int256 denominator = totalAllocatedTokensRounded *
↳ int256(lastPrices[_protocolId]) * 100; // * 100 cause perfFee is in
↳ percentages

    if (totalAllocatedTokensRounded == 0) {
        rewardPerLockedToken[rebalancingPeriod][_protocolId] = 0;
    } else {
        rewardPerLockedToken[rebalancingPeriod][_protocolId] = nominator /
↳ denominator;
    }

    lastPrices[_protocolId] = currentPrice;
}
```

price() is Provider's exchangeRate():

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L381-L390>

```
/// @notice Get price for underlying protocol
/// @param _protocolNum Protocol number linked to an underlying protocol e.g
↳ compound_usdc_01
/// @return protocolPrice Price per lp token
function price(uint256 _protocolNum) public view returns (uint256) {
```



```

IController.ProtocolInfoS memory protocol = controller.getProtocolInfo(
    vaultNumber,
    _protocolNum
);
return IProvider(protocol.provider).exchangeRate(protocol.LPToken);
}

```

IdleProvider's exchangeRate() is scaled with underlying token decimals:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/IdleProvider.sol#L113-L118>

```

/// @notice Exchange rate of underlying protocol token
/// @param _iToken Address of protocol LP Token eg yUSDC
/// @return price of LP token
function exchangeRate(address _iToken) public view override returns (uint256) {
    return IIdle(_iToken).tokenPrice();
}

```

https://github.com/Idle-Labs/idle-contracts/blob/develop/contracts/IdleTokenV3_1.sol#L240-L245

```

/**
 * IdleToken price calculation, in underlying
 *
 * @return : price in underlying token
 */
function tokenPrice() external view returns (uint256) {}

```

Tool used

Manual Review

Recommendation

Consider enhancing performance of the rewards calculation and Game's baskets[_basketId].totalUnRedeemedRewards, for example:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L46-L47>

```

uint256 public uScale;

uint256 public minimumPull;

+ uint256 public BASE_SCALE = 1e18;

```



<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L220-L245>

```
    /// @notice Stores the historical price and the reward per rounded locked
    ↪ token, ignoring decimals.
    /// @dev formula yield protocol i at time t:  $y(it) = (P(it) - P(it-1)) /$ 
    ↪  $P(it-1)$ .
    /// @dev formula rewardPerLockedToken for protocol i at time t:  $r(it) = y(it)$ 
    ↪  $* TVL(t) * perfFee(t) / totalLockedTokens(t)$ 
    /// @dev later, when the total rewards are calculated for a game player we
    ↪ multiply this ( $r(it)$ ) by the locked tokens on protocol i at time t
    /// @param _totalUnderlying Totalunderlying = TotalUnderlyingInProtocols -
    ↪ BalanceVault.
    /// @param _protocolId Protocol id number.
    function storePriceAndRewards(uint256 _totalUnderlying, uint256 _protocolId)
    ↪ internal {
        uint256 currentPrice = price(_protocolId);
        if (lastPrices[_protocolId] == 0) {
            lastPrices[_protocolId] = currentPrice;
            return;
        }

        int256 priceDiff = int256(currentPrice - lastPrices[_protocolId]);
        - int256 nominator = (int256(_totalUnderlying * performanceFee) * priceDiff);
        + int256 nominator = (int256(_totalUnderlying * performanceFee) * priceDiff *
    ↪ BASE_SCALE);
        int256 totalAllocatedTokensRounded = totalAllocatedTokens / 1E18;
        int256 denominator = totalAllocatedTokensRounded *
    ↪ int256(lastPrices[_protocolId]) * 100; // * 100 cause perfFee is in
    ↪ percentages

        if (totalAllocatedTokensRounded == 0) {
            rewardPerLockedToken[rebalancingPeriod][_protocolId] = 0;
        } else {
            rewardPerLockedToken[rebalancingPeriod][_protocolId] = nominator /
    ↪ denominator;
        }

        lastPrices[_protocolId] = currentPrice;
    }
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L69-L70>

```
// percentage of tokens that will be sold at negative rewards
uint256 internal negativeRewardFactor;
```



```
+ uint256 public BASE_SCALE = 1e18;
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L296-L306>

```
function redeemNegativeRewards(
    uint256 _basketId,
    uint256 _unlockedTokens
) internal returns (uint256) {
    int256 unredeemedRewards = baskets[_basketId].totalUnRedeemedRewards;
    if (unredeemedRewards > negativeRewardThreshold) return 0;

    uint256 tokensToBurn = (uint(-unredeemedRewards) * negativeRewardFactor) /
↳ 100;
    tokensToBurn = tokensToBurn < _unlockedTokens ? tokensToBurn :
↳ _unlockedTokens;

- baskets[_basketId].totalUnRedeemedRewards += int((tokensToBurn * 100) /
↳ negativeRewardFactor);
+ baskets[_basketId].totalUnRedeemedRewards += int((tokensToBurn * 100 *
↳ BASE_SCALE) / negativeRewardFactor);
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L542-L553>

```
/// @notice redeem funds from basket in the game.
/// @dev makes a call to the vault to make the actual transfer because the
↳ vault holds the funds.
/// @param _basketId Basket ID (tokenId) in the BasketToken (NFT) contract.
function redeemRewards(uint256 _basketId) external onlyBasketOwner(_basketId) {
- int256 amount = baskets[_basketId].totalUnRedeemedRewards;
+ int256 amount = baskets[_basketId].totalUnRedeemedRewards / BASE_SCALE;
    require(amount > 0, "Nothing to claim");

    baskets[_basketId].totalRedeemedRewards += amount;
    baskets[_basketId].totalUnRedeemedRewards = 0;

    IVault(homeVault).redeemRewardsGame(uint256(amount), msg.sender);
}
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L19-L32>

```
struct Basket {
    // the vault number for which this Basket was created
    uint256 vaultNumber;
    // last period when this Basket got rebalanced
```



```
uint256 lastRebalancingPeriod;  
// nr of total allocated tokens  
int256 nrOfAllocatedTokens;  
// total build up rewards  
- int256 totalUnRedeemedRewards;  
+ int256 totalUnRedeemedRewards; // in {underlying + 18} decimals precision  
// total redeemed rewards  
int256 totalRedeemedRewards;  
// (basket => vaultNumber => chainId => allocation)  
mapping(uint256 => mapping(uint256 => int256)) allocations;  
}
```

Discussion

sjoerdsommen

Possibly a high severity

hrishibhat

Given that this applies only for tokens with lower decimals considering this a valid medium



Issue M-27: IdleProvider overstates balanceUnderlying() and understates calcShares()

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/206>

Found by

hyh

Summary

IdleProvider's exchangeRate() use Idle's tokenPrice() which doesn't include performance based fees, so balanceUnderlying() and calcShares() report figures that may not match with what Vault will effectively obtain on interaction with Idle.

Vulnerability Detail

Idle has abstract tokenPrice() and user-dependent tokenPriceWithFee(), which takes into account performance fees.

tokenPrice(), which do not have user information and report price excluding any performance fees, isn't a correct way to estimate current balance with Idle pool and the quantity of shares needed to withdraw a particular amount of underlying, tokenPriceWithFee() is.

Impact

IdleProvider overstates balance of underlying and understates the number of shares needed to withdraw given amount of underlying.

This will skew rebalancing as result of it will be less than requested whenever Idle strategy obtained a profit and will have performance fees deducted on withdrawal. This will bias the distribution vs expected and can lead to protocol-wide losses accumulated over time.

Also, Derby internal performance metric is inflated this way for Idle pools as the same no-fee rate is used to gauge it.

Code Snippet

IdleProvider returns tokenPrice() in exchangeRate():

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/IdleProvider.sol#L113-L118>



```

/// @notice Exchange rate of underlying protocol token
/// @param _iToken Address of protocol LP Token eg yUSDC
/// @return price of LP token
function exchangeRate(address _iToken) public view override returns (uint256) {
    return IIIdle(_iToken).tokenPrice();
}

```

It is then used for balance of underlying and shares per amount of underlying estimations:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/IdleProvider.sol#L79-L103>

```

/// @notice Get balance from address in underlying token
/// @dev balance = poolvalue * shares / totalsupply
/// @param _address Address to request balance from, most likely an Vault
/// @param _iToken Address of protocol LP Token eg cUSDC
/// @return balance in underlying token
function balanceUnderlying(
    address _address,
    address _iToken
) public view override returns (uint256) {
    uint256 balanceShares = balance(_address, _iToken);
    uint256 price = exchangeRate(_iToken);
    uint256 decimals = IERC20Metadata(IIIdle(_iToken).token()).decimals();
    return (balanceShares * price) / 10 ** decimals;
}

/// @notice Calculates how many shares are equal to the amount
/// @dev shares = totalsupply * balance / poolvalue
/// @param _amount Amount in underlying token e.g USDC
/// @param _iToken Address of protocol LP Token eg cUSDC
/// @return number of shares i.e LP tokens
function calcShares(uint256 _amount, address _iToken) external view override
    ↪ returns (uint256) {
    uint256 decimals = IERC20Metadata(IIIdle(_iToken).token()).decimals();
    uint256 shares = (_amount * (10 ** decimals)) / exchangeRate(_iToken);
    return shares;
}

```

However, Idle has substantial enough performance fees (10% and higher), which are removed from a profit generated for a particular user.

<https://docs.idle.finance/developers/best-yield/methods/tokenprice>

<https://docs.idle.finance/developers/best-yield/methods/tokenpricewithfee>



This method returns \$IDLE token price for a specific user considering fees, in
↳ underlying. This is useful when you need to redeem exactly X amount of
↳ underlying tokens.

This way tokenPrice() isn't a correct way for a Vault to estimate what will be obtained from Idle as it overstates share price not including the fees due.

balanceUnderlying() and calcShares() are used in Vault rebalancing, for example:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L178-L192>

```
function rebalanceCheckProtocols(
    uint256 _newTotalUnderlying
) internal returns (uint256[] memory) {
    uint256[] memory protocolToDeposit = new
    ↳ uint[](controller.latestProtocolId(vaultNumber));
    uint256 latestID = controller.latestProtocolId(vaultNumber);
    for (uint i = 0; i < latestID; i++) {
        bool isBlacklisted = controller.getProtocolBlacklist(vaultNumber, i);

        storePriceAndRewards(_newTotalUnderlying, i);

        if (isBlacklisted) continue;
        setAllocation(i);

        int256 amountToProtocol = calcAmountToProtocol(_newTotalUnderlying, i);
        uint256 currentBalance = balanceUnderlying(i);
```

exchangeRate() is also used in Vault's price() for measuring performance, which is overstated this way for Idle:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L381-L390>

```
/// @notice Get price for underlying protocol
/// @param _protocolNum Protocol number linked to an underlying protocol e.g
    ↳ compound_usdc_01
/// @return protocolPrice Price per lp token
function price(uint256 _protocolNum) public view returns (uint256) {
    IController.ProtocolInfoS memory protocol = controller.getProtocolInfo(
        vaultNumber,
        _protocolNum
    );
    return IProvider(protocol.provider).exchangeRate(protocol.LPToken);
}
```



Tool used

Manual Review

Recommendation

Consider using `tokenPriceWithFee(msg.sender)` for expected amounts:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/IdleProvider.sol#L113-L118>

```
/// @notice Exchange rate of underlying protocol token
/// @param _iToken Address of protocol LP Token eg yUSDC
/// @return price of LP token
function exchangeRate(address _iToken) public view override returns (uint256) {
-   return IIdle(_iToken).tokenPrice();
+   return IIdle(_iToken).tokenPriceWithFee(msg.sender);
}
```

Discussion

sjoerdsommen

Will slightly skew the totalUnderlying, no funds are lost.



Issue M-28: Malicious users could set allocations to a blacklist Protocol and break the rebalancing logic

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/192>

Found by

Met, bin2chen, immeas, Ch_301, saidam017

Summary

game.sol pushes deltaAllocations to vaults by pushAllocationsToVaults() and it deletes all the value of the deltas

```
vaults[_vaultNumber].deltaAllocationProtocol[_chainId][i] = 0;
```

Vulnerability Detail

Malicious users could set allocations to a blacklist Protocol. If only one of the Baskets has a non-zero value to a **Protocol on blacklist**

receiveProtocolAllocations() will revert receiveProtocolAllocations().receiveProtocolAllocationsInt().setDeltaAllocationsInt()

```
function setDeltaAllocationsInt(uint256 _protocolNum, int256 _allocation)
↳ internal {
    require(!controller.getProtocolBlacklist(vaultNumber, _protocolNum), "Protocol
↳ on blacklist");
    deltaAllocations[_protocolNum] += _allocation;
    deltaAllocatedTokens += _allocation;
}
```

and You won't be able to execute rebalance()

Impact

The guardian isn't able to restart the protocol manually. game.sol loses the value of the deltas. The whole system is down.

Code Snippet

Tool used

Manual Review



Recommendation

You should check if the Protocol on the blacklist when Game players
`rebalanceBasket()`

Discussion

sjoerdsommen

duplicate with #168



Issue M-29: Manipulate Allocations from game using flashloans

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/172>

Found by

oot2k

Summary

A user can use flashloans to leverage the amount of derby token dedicated towards a certain protocol.

Vulnerability Detail

Consider the following scenario: A malicious user wants to manipulate DeltaAllocations to change the weight of some protocol inside vault. This works because the user can call `pushAllocationsToController` and `pushAllocationsToVaults` in the same transaction. <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L424> <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L465>

1. They take a flashloan of USDC
2. Swap the USDC into derby token on some dex
3. Rebalance there basket using these tokens
4. Call `pushAllocationsToController`
5. Call `pushAllocationsToVaults`
6. Rebalance there basket to get tokens back
7. Swap tokens to USDC
8. pay back loan

Impact

A User can use this to stake tokens into bad performing protocols, leading to possible lose of funds or unexpected behavior.

Code Snippet

Tool used

Manual Review



Recommendation

do not allow to call `pushAllocationsToController` and `pushAllocationsToVaults` in same transaction.



Issue M-30: Funds can be frozen on protocol blacklisting

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/156>

Found by

Ruhum, rvierdiiev, Ch_301, hyh, csanuragjain

Summary

When a protocol is blacklisted by Vault's `blacklistProtocol()` it isn't controlled for full withdrawal of user funds and also there is no rewards claiming performed.

This way both

- 1) locked funds, i.e. a share of protocol deposit that is generally available, but cannot be withdrawn at the moment due to liquidity situation, and
- 2) accumulated, but not yet claimed rewards

are frozen within the blacklisted protocol.

Vulnerability Detail

`blacklistProtocol()` can be routinely run for a protocol that still has some allocation, but claiming isn't called there and `withdrawFromProtocol()` performed can end up withdrawing only partial amount of funds (for example because of lending liquidity squeeze).

Protocol blacklisting can be urgent, in which case leaving some funds with the protocol can be somewhat unavoidable, and can be not urgent, in which case locking any meaningful funds with the protocol isn't desirable. Now there is no control lever to distinguish between these scenarios, so the funds can be locked when it can be avoided.

Impact

Funds attributed to Vault LPs are frozen when a protocol is blacklisted: both reward funds and invested funds that aren't lost, but cannot be withdrawn at the moment due to current liquidity situation of the protocol.

Code Snippet

`blacklistProtocol()` performs no rewards claiming and will proceed when withdrawal wasn't full:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L475-L483>



```

/// @notice The DAO should be able to blacklist protocols, the funds should be
↳ sent to the vault.
/// @param _protocolNum Protocol number linked to an underlying vault e.g
↳ compound_usdc_01
function blacklistProtocol(uint256 _protocolNum) external onlyGuardian {
    uint256 balanceProtocol = balanceUnderlying(_protocolNum);
    currentAllocations[_protocolNum] = 0;
    controller.setProtocolBlacklist(vaultNumber, _protocolNum);
    savedTotalUnderlying -= balanceProtocol;
    withdrawFromProtocol(_protocolNum, balanceProtocol);
}

```

Tool used

Manual Review

Recommendation

Consider performing claiming for the blacklisted protocol, for example replace `currentAllocations[_protocolNum] = 0` with logic similar to `claimTokens()`:

```

if (currentAllocations[_protocolNum] > 0) {
    bool claim = controller.claim(vaultNumber, _protocolNum);
    if (claim) {
        address govToken = controller.getGovToken(vaultNumber, _protocolNum);
        uint256 tokenBalance = IERC20(govToken).balanceOf(address(this));
        Swap.swapTokensMulti(
            Swap.SwapInOut(tokenBalance, govToken, address(vaultCurrency)),
            controller.getUniswapParams(),
            false
        );
    }
    currentAllocations[_protocolNum] = 0;
}

```

Also, consider introducing an option to revert when funds actually withdrawn from `withdrawFromProtocol()` are lesser than `balanceProtocol` by more than a threshold, i.e. the flag will block blacklisting if too much funds are being left in the protocol.



Issue M-31: minimumPull Vault parameter cannot be adjusted

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/154>

Found by

hyh

Summary

`minimumPull` is hardcoded to be 10^6 in Vault and can't be reset thereafter.

If vault token has low decimals and high enough value, for example WBTC having 8 decimals (<https://etherscan.io/token/0x2260fac5e5542a773aa44fbcfedf7c193bc2c599#code>), `minimumPull` can block funds pulling when it is needed for Vault withdrawals.

Vulnerability Detail

Suppose BTC is USD 300k and a new Vault has USD 30k AUM. If vault token is WBTC the value of `minimumPull` = 0.01 WBTC will be around USD 3000, which is 10% of the Vault AUM.

In this case oftentimes withdrawals requested will not be that high, so `minimumPull` check can block substantial share of protocol withdrawals when it is needed to cover Vault LP withdrawal queue.

Impact

Vault withdrawal requests cannot be processed for an arbitrary time while `reservedFunds` and Vault balance difference is lower than threshold, i.e. LP funds will be frozen in the Vault.

Code Snippet

`minimumPull` is hardcoded to 10^6 on construction:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L90-L105>

```
constructor(  
    uint256 _vaultNumber,  
    address _dao,  
    address _controller,  
    address _vaultCurrency,
```




```

    uint256 _uScale
) {
    controller = IController(_controller);
    vaultCurrency = IERC20(_vaultCurrency);

    vaultNumber = _vaultNumber;
    dao = _dao;
    uScale = _uScale;
    lastTimeStamp = block.timestamp;
    minimumPull = 1_000_000;
}

```

For example, `maxDivergenceWithdraws` is hardcoded to be 10^6 in `MainVault` too:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L50-L68>

```

constructor(
    ...
)
    VaultToken(_name, _symbol, _decimals)
    Vault(_vaultNumber, _dao, _controller, _vaultCurrency, _uScale)
{
    ...
    maxDivergenceWithdraws = 1_000_000;
}

```

But it's resettable:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L418-L420>

```

function setMaxDivergence(uint256 _maxDivergence) external onlyDao {
    maxDivergenceWithdraws = _maxDivergence;
}

```

But `minimumPull` can't be reset.

It is impossible to withdraw less than `minimumPull` from protocols:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L111-L127>

```

function pullFunds(uint256 _value) internal {
    uint256 latestID = controller.latestProtocolId(vaultNumber);
    for (uint i = 0; i < latestID; i++) {
        if (currentAllocations[i] == 0) continue;
    }
}

```



```

uint256 shortage = _value - vaultCurrency.balanceOf(address(this));
uint256 balanceProtocol = balanceUnderlying(i);

uint256 amountToWithdraw = shortage > balanceProtocol ? balanceProtocol :
↳ shortage;
savedTotalUnderlying -= amountToWithdraw;

if (amountToWithdraw < minimumPull) break;
withdrawFromProtocol(i, amountToWithdraw);

if (_value <= vaultCurrency.balanceOf(address(this))) break;
}
}

```

When Vault balance is lacking the LP withdrawals are blocked:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L131-L144>

```

function withdraw(
    uint256 _amount,
    address _receiver,
    address _owner
) external nonReentrant onlyWhenVaultIsOn returns (uint256 value) {
    value = (_amount * exchangeRate) / (10 ** decimals());

    require(value > 0, "!value");

    require(getVaultBalance() - reservedFunds >= value, "!funds");

    _burn(msg.sender, _amount);
    transferFunds(_receiver, value);
}

```

Tool used

Manual Review

Recommendation

Consider introducing a setter for `minimumPull`, so it can be reset to a relevant level with regard to vault token decimals and its market value:

```

function setMinimumPull(uint256 _minimumPull) external onlyDao {
    minimumPull = _minimumPull;
}

```



This will also allow to dynamically adjust the Vaults thereafter (say 1e6 is ok as threshold while WBTC is 20k, but becomes prohibiting when it is 200k).



Issue M-32: User should not receive rewards for the rebalance period, when protocol was blacklisted, because of unpredicted behaviour of protocol price

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/145>

Found by

Bauer, rvierdiiev, immeas, Ch_301, evan

Summary

User should not receive rewards for the rebalance period, when protocol was blacklisted, because of unpredicted behaviour of protocol price.

Vulnerability Detail

When user allocates derby tokens to some underlying protocol, he receive rewards according to the exchange price of that protocols token. This reward can be positive or negative. Rewards of protocol are set to `Game` contract inside `settleRewards` function and they are accumulated for user, once he calls `rebalanceBasket`.

Let's check how they are calculated. <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L226-L245>

```
function storePriceAndRewards(uint256 _totalUnderlying, uint256 _protocolId)
↳ internal {
    uint256 currentPrice = price(_protocolId);
    if (lastPrices[_protocolId] == 0) {
        lastPrices[_protocolId] = currentPrice;
        return;
    }

    int256 priceDiff = int256(currentPrice - lastPrices[_protocolId]);
    int256 nominator = (int256(_totalUnderlying * performanceFee) * priceDiff);
    int256 totalAllocatedTokensRounded = totalAllocatedTokens / 1E18;
    int256 denominator = totalAllocatedTokensRounded *
↳ int256(lastPrices[_protocolId]) * 100; // * 100 cause perfFee is in
↳ percentages

    if (totalAllocatedTokensRounded == 0) {
        rewardPerLockedToken[rebalancingPeriod][_protocolId] = 0;
    } else {
```



```

        rewardPerLockedToken[rebalancingPeriod][_protocolId] = nominator /
↪ denominator;
    }

    lastPrices[_protocolId] = currentPrice;
}

```

Every time, previous price of protocol is compared with current price.

In case if some protocol is hacked, there is `Vault.blacklistProtocol` function, that should withdraw reserves from protocol and mark it as blacklisted. The problem is that because of the hack it's not possible to determine what will happen with exchange rate of protocol. It can be 0, or it can be very small or it can be high for any reasons. But protocol still accrues rewards per token for protocol, even that it is blacklisted. Because of that, user that allocated to that protocol can face with accruing very big negative or positive rewards. Both this cases are bad.

So i believe that in case if protocol is blacklisted, it's better to set rewards as 0 for it.

Example. 1.User allocated 100 derby tokens for protocol A 2.Before `Vault.rebalance` call, protocol A was hacked which made it `exchangeRate` to be not real. 3.Derby team has blacklisted that protocol A. 4.`Vault.rebalance` is called which used new(incorrect) `exchangeRate` of protocol A in order to calculate `rewardPerLockedToken` 5.When user calls `rebalance basket` next time, his rewards are accumulated with extremely high/low value.

Impact

User's rewards calculation is unpredictable.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

In case if protocol is blacklisted, then set `rewardPerLockedToken` to 0 inside `storePriceAndRewards` function.



Issue M-33: The protocol could not handle multiple vaults correctly

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/140>

Found by

SPYBOY, Ch_301

Summary

The protocol needs to handle multiple vaults correctly. If there are three vaults (e.g. USDC, USDT, DAI) the protocol needs to rebalance them all without any problems

Vulnerability Detail

The protocol needs to invoke `pushAllocationsToController()` every `rebalanceInterval` to push **totalDeltaAllocations** from **Game** to **xChainController**.

`pushAllocationsToController()` invoke `rebalanceNeeded()` to check if a rebalance is needed based on the set interval and it uses the state variable `lastTimeStamp` to do the calculations

```
function rebalanceNeeded() public view returns (bool) {
    return (block.timestamp - lastTimeStamp) > rebalanceInterval || msg.sender ==
    ↪ guardian;
}
```

But in the first invoking (for USDC vault) of `pushAllocationsToController()` it will update the state variable `lastTimeStamp` to the current `block.timestamp`

```
lastTimeStamp = block.timestamp;
```

Now when you invoke (for DAI vault) `pushAllocationsToController()`. It will revert because of

```
require(rebalanceNeeded(), "No rebalance needed");
```

So if the protocol has two vaults or more (USDC, USDT, DAI) you can only do one rebalance every `rebalanceInterval`

Impact

- The protocol could not handle multiple vaults correctly



- Both Users and Game players will lose funds because the MainVault will not rebalance the protocols at the right time with the right values

Code Snippet

```
function pushAllocationsToController(uint256 _vaultNumber) external payable {
    require(rebalanceNeeded(), "No rebalance needed");
    for (uint k = 0; k < chainIds.length; k++) {
        require(
            getVaultAddress(_vaultNumber, chainIds[k]) != address(0),
            "Game: not a valid vaultnumber"
        );
        require(
            !isXChainRebalancing[_vaultNumber][chainIds[k]],
            "Game: vault is already rebalancing"
        );
        isXChainRebalancing[_vaultNumber][chainIds[k]] = true;
    }
}
```

Tool used

Manual Review

Recommendation

Keep tracking the lastTimeStamp for every _vaultNumber by using an array



Issue M-34: There is no price conversion between vault token and provider underlying token amounts in withdrawFromProtocol

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/135>

Found by

Met, c7e7eff, immeas, rvierdiev, gogo, hyh, tives, HonorLt, Bobface, Jeiwan

Summary

Vault's `withdrawFromProtocol()` use its `vaultCurrency` based `_amount` argument without price conversion into Provider's underlying token, which is expected by the `calcShares()` call. As stablecoin prices can diverge, even massively, it isn't appropriate to assume price equality and use one amount instead of another.

Currently the final withdrawn amount is converted into `vaultCurrency`, but the withdrawal request is being made as if the `vaultCurrency` and Provider's underlying token are worth exactly the same, which almost always isn't the case.

Vulnerability Detail

Suppose `vaultCurrency` is USDC, while Provider's underlying currency is USDT, Provider is AaveProvider.

Suppose USDT is went through a major regulatory scrutiny and it is now priced at 0.8 USD, but this is a locally stable situation and the major USDT Vaults now offer quite attractive returns, so Aave USDT was whitelisted.

`withdrawFromProtocol()` being called with 10^{12} amount, which is a request to withdraw 10^6 USDC, while AaveProvider's `calcShares()` will be called without price conversion, i.e. it will treat USDC amount supplied as USDT amount as it's the underlying token. Upon withdrawal the result will be converted in `vaultCurrency` USDC, always resulting in 20% less value than it was requested.

Impact

Withdrawal from a protocol is a base functionality of the Vault and it will malfunction whenever vault currency and Provider's underlying currency aren't equally priced, which is the case along with market volatility.

Net impact is ongoing Vault misbalancing, mostly mild, but sometimes substantial, which will alter the effective distribution vs desired and shift actual results from expected, leading to losses for protocol depositors.



Code Snippet

withdrawFromProtocol's input argument `_amount` is in `vaultCurrency`, and it is provided after decimals rebase, but without any cross-price conversion to `calcShares()`:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L303-L316>

```
/// @notice Withdraw amount from underlying protocol
/// @dev shares = amount / PricePerShare
/// @param _protocolNum Protocol number linked to an underlying protocol e.g
↳ compound_usdc_01
/// @param _amount in VaultCurrency to withdraw
function withdrawFromProtocol(uint256 _protocolNum, uint256 _amount) internal {
    if (_amount <= 0) return;
    IController.ProtocolInfoS memory protocol = controller.getProtocolInfo(
        vaultNumber,
        _protocolNum
    );

    _amount = (_amount * protocol.uScale) / uScale;
    uint256 shares = IProvider(protocol.provider).calcShares(_amount,
↳ protocol.LPToken);
    uint256 balance = IProvider(protocol.provider).balance(address(this),
↳ protocol.LPToken);
```

All `calcShares()` methods expect `_amount` to be in the Provider underlying token, not in `vaultCurrency`:

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/AaveProvider.sol#L88-L96>

```
/// @notice Calculates how many shares are equal to the amount
/// @dev Aave exchangeRate is 1
/// @param _amount Amount in underlying token e.g USDC
/// @param _aToken Address of protocol LP Token eg aUSDC
/// @return number of shares i.e LP tokens
function calcShares(uint256 _amount, address _aToken) external view override
↳ returns (uint256) {
    uint256 shares = _amount / exchangeRate(_aToken);
    return shares;
}
```

...

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Providers/CompoundProvider.sol#L102-L111>



```

/// @notice Calculates how many shares are equal to the amount
/// @dev returned price from compound is scaled
↪ https://compound.finance/docs/ctokens#exchange-rate
/// @param _amount Amount in underlying token e.g USDC
/// @param _cToken Address of protocol LP Token eg cUSDC
/// @return number of shares i.e LP tokens
function calcShares(uint256 _amount, address _cToken) external view override
↪ returns (uint256) {
    uint256 decimals = IERC20Metadata(ICToken(_cToken).underlying()).decimals();
    uint256 shares = (_amount * (10 ** (10 + decimals))) / exchangeRate(_cToken);
    return shares;
}

```

Cross-currency prices can vary even in the stablecoins case and whenever rebalancing coincides with cross-rate volatility spike the effective Vault weights will be disturbed.

Tool used

Manual Review

Recommendation

Consider correcting for the current vault currency to underlying token exchange rate, for example:

```

IController.UniswapParams uniParams = controller.getUniswapParams();
IController.UniswapParams curveParams = controller.getCurveParams();
underlyingAmount = Swap.amountOutMultiSwap(
    Swap.SwapInOut(_amount, address(vaultCurrency), protocol.underlying),
    uniParams.quoter,
    0
);
underlyingAmount = (underlyingAmount * (10000 + curveParams.poolFee)) / 10000;

```

and using the underlyingAmount for shares estimation via calcShares().

Discussion

sjøerdsommen

Duplicate with #314



Issue M-35: Vault.blacklistProtocol can revert in emergency

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/128>

Found by

rvierdiiev

Summary

Vault.blacklistProtocol can revert in emergency, because it tries to withdraw underlying balance from protocol, which can revert for many reasons after it's hacked or paused.

Vulnerability Detail

Vault.blacklistProtocol is created for emergency cases and it's needed to remove protocol from protocols that allowed to deposit. <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L477-L483>

```
function blacklistProtocol(uint256 _protocolNum) external onlyGuardian {
    uint256 balanceProtocol = balanceUnderlying(_protocolNum);
    currentAllocations[_protocolNum] = 0;
    controller.setProtocolBlacklist(vaultNumber, _protocolNum);
    savedTotalUnderlying -= balanceProtocol;
    withdrawFromProtocol(_protocolNum, balanceProtocol);
}
```

The problem is that this function is trying to withdraw all balance from protocol. This can create problems as in case of hack, attacker can steal funds, pause protocol and any other things that can make withdrawFromProtocol function to revert. Because of that it will be not possible to add protocol to blacklist and as result system will stop working correctly.

Impact

Hacked or paused protocol can't be set to blacklist.

Code Snippet

Provided above



Tool used

Manual Review

Recommendation

Provide `needToWithdraw` param to the `blacklistProtocol` function. In case if it's safe to withdraw, then withdraw, otherwise, just set protocol as blacklisted. Also you can call function with `true` param again, once it's safe to withdraw. Example of hack situation flow: 1.underlying vault is hacked 2.you call `setProtocolBlacklist("vault", false)` which blacklists vault 3.in next tx you call `setProtocolBlacklist("vault", true)` and tries to withdraw

Discussion

sjoerdsommen

Disagree with severity because it's unlikely to result in (extra) loss of funds



Issue M-36: exchangeRate is not up to date in case if vault that is off is changed to on

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/126>

Found by

rvierdiiev

Summary

exchangeRate is not up to date in case if vault that is off is changed to on. Depositors and withdrawers will use old exchangeRate that will be less than in reality.

Vulnerability Detail

In case if vault has no allocations, then it is switched off. That means that users can't deposit/withdraw from vault anymore. After it was detected that vault is off or on, then `sendFeedbackToVault` is called.

In case if vault is off, that means that actually it doesn't have underlying inside providers anymore, so there is no reason send exchangeRate to it.

exchangeRate is very important param as it is used in share calculations when user deposits/withdraws. That's why it's important that when vault is ON again after it was OFF, exchangeRate is up to date.

In case if vault is off, then exchange rate is not provided to it. Then when after few rebalancing period, some allocation was provided for the vault and it was marked as ON again and `sendFeedbackToVault` is called, then users can deposit/withdraw again. But the problem is that exchangeRate is old there and is likely less than it should be, so depositors receive more shares and withdrawers receive less funds.

Example. 1.At cycle 10 exchangeRate is 1.1. 2.Vault has no more allocations and it's switched off. 3.At cycle 15 new allocations were send to vault. So it's switched on. Current exchangeRate is 1.2. 4.Depositors now can deposit, but exchangeRate inside this vault is not fresh, it's from 10th cycle. 5.So some depositors had chance to receive more shares and some withdrawers lost some funds.

I am not sure about severity here, think it's medium because switching on/off can not be often.

Impact

Incorrect shares calculation



Code Snippet

Provided above

Tool used

Manual Review

Recommendation

When you set vault to on state, you need to send current exchangeRate as well.



Issue M-37: Vault.claim should be called before `pushTotalUnderlying`

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/123>

Found by

rvierdiiev, Ch_301

Summary

`Vault.claim` should be called before `pushTotalUnderlyingToController`.

Vulnerability Detail

Every cycle, each vault should send its total underlying amount to `xController`, so `xController` can calculate according to new allocations, which amount of underlying vault should receive from/send to `xController`. Also it calculates `exchangeRate` that will be used for new depositors/withdrawers.

`MainVault.pushTotalUnderlyingToController` is responsible for sending underlying amount. It first calls `setTotalUnderlying` function, which will calculate amount of underlying inside all active providers. And then it will calculate total amount by adding funds controlled by vault without reserved funds amount. `uint256 underlying = savedTotalUnderlying + getVaultBalance() - reservedFunds;` Then this amount is sent to `xController`.

Now let's check another function `Vault.claimTokens` which is called inside rebalance function. The purpose of this function is to receive earned rewards tokens from providers and then swap them to vaultCurrency token.

Of course, this increases `getVaultBalance()`.

So the main point of this bug is that `Vault.claim` should be called inside `pushTotalUnderlyingToController` as this is also funds that should be then distributed among vaults according to allocations and it should also increase `exchangeRate`.

Impact

Underlying amount sent to `xController` is not accurate, actually vault can have more funds. Exchange rate is also not accurate.

Code Snippet

Provided above



Tool used

Manual Review

Recommendation

Call `Vault.claim` inside `pushTotalUnderlyingToController`.



Issue M-38: Rewards per locked token are calculated incorrectly

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/120>

Found by

rvierdiiev

Summary

Rewards per locked token are calculated incorrectly as they are calculated, when at the allocation time, but should be calculated at the end of rebalancing period.

Vulnerability Detail

The purpose of Game is to allow user to make allocations into different providers. Depending on the price of providers LP and staked amount of derby token, player receive rewards for the rebalancing period. So actually user should allocate amount, then rebalancing period is started and according to the allocations of players different providers are funded. Then at the end of rebalance period, rewards should be calculated that players were able to earn by their allocations.

The problem is that protocol currently implements this in different way. Rewards are calculated, once vault received new allocations. So actually this allocations even didn't have time to work in order to calculate earning.

So once allocations are received from the game and XController provided needed underlying, then `Vault.rebalance` is called. This function will also call `rebalanceCheckProtocols`, which then calls storePriceAndRewards function. This function is responsible for calculating rewards per locked token for provider.

And as you can see it is called too early, as allocations and underlying for them just arrived and they are not even deposited yet into providers. So you shouldn't calculate earnings for that allocations yet, you need to wait till end of rebalance period.

End of rebalance period is when `MainVault.sendRewardsToGame` is called. So i believe, rewards should be calculated exactly at this function.

Impact

Rewards per locked token are calculated incorrectly.

Code Snippet

Provided above



Tool used

Manual Review

Recommendation

Calculate rewards inside `MainVault.sendRewardsToGame` function.



Issue M-39: Game doesn't accrued rewards for previous rebalance period in case if rebalanceBasket is called in next period

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/119>

Found by

XKET, rvierdiev, bin2chen, tallo, chainNue

Summary

Game doesn't accrued rewards for previous rebalance period in case if rebalanceBasket is called in next period. Because of that user do not receive rewards for the previous period and in case if he calls rebalanceBasket each rebalance period, he will receive rewards only for last one.

Vulnerability Detail

When Game.rebalanceBasket is called, then basket rewards are accrued by calling addToTotalRewards function. <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Game.sol#L368-L401>

```
function addToTotalRewards(uint256 _basketId) internal
↳ onlyBasketOwner(_basketId) {
    if (baskets[_basketId].nrOfAllocatedTokens == 0) return;

    uint256 vaultNum = baskets[_basketId].vaultNumber;
    uint256 currentRebalancingPeriod = vaults[vaultNum].rebalancingPeriod;
    uint256 lastRebalancingPeriod = baskets[_basketId].lastRebalancingPeriod;

    if (currentRebalancingPeriod <= lastRebalancingPeriod) return;

    for (uint k = 0; k < chainIds.length; k++) {
        uint32 chain = chainIds[k];
        uint256 latestProtocol = latestProtocolId[chain];
        for (uint i = 0; i < latestProtocol; i++) {
            int256 allocation = basketAllocationInProtocol(_basketId, chain, i) / 1E18;
            if (allocation == 0) continue;

            int256 lastRebalanceReward = getRewardsPerLockedToken(
```



```

        vaultNum,
        chain,
        lastRebalancingPeriod,
        i
    );
    int256 currentReward = getRewardsPerLockedToken(
        vaultNum,
        chain,
        currentRebalancingPeriod,
        i
    );
    baskets[_basketId].totalUnRedeemedRewards +=
        (currentReward - lastRebalanceReward) *
        allocation;
    }
}
}

```

This function allows user to accrue rewards only when `currentRebalancingPeriod > lastRebalancingPeriod`. When user allocates, he allocates for the next period. And `lastRebalancingPeriod` is changed after `addToTotalRewards` is called, so after rewards for previous period accrued. And when allocations are sent to the `xController`, then new rebalance period is started. So actually rewards accruing for period that user allocated for is started once `pushAllocationsToController` is called. And at this point `currentRebalancingPeriod == lastRebalancingPeriod` which means that if user will call `rebalanceBasket` for next period, the rewards will not be accrued for him, but `lastRebalancingPeriod` will be incremented. So actually he will not receive rewards for previous period.

Example. 1.`currentRebalancingPeriod` is 10. 2.user calls `rebalanceBasket` with new allocation and `lastRebalancingPeriod` is set to 11 for him. 3.`pushAllocationsToController` is called, so `currentRebalancingPeriod` becomes 11. 4.`settleRewards` is called, so rewards for the 11th cycle are accrued. 5.now user can call `rebalanceBasket` for the next 12th cycle. `addToTotalRewards` is called, but `currentRebalancingPeriod == lastRebalancingPeriod == 11`, so rewards were not accrued for 11th cycle 6.new allocations is saved and `lastRebalancingPeriod` becomes 12. 7.the loop continues and every time when user allocates for next rewards his `lastRebalancingPeriod` is increased, but rewards are not added. 8.user will receive his rewards for previous cycle, only if he skip 1 rebalance period(he doesn't allocate on that period).

As you can see this is very serious bug. Because of that, player that wants to adjust his allocation every rebalance period will loose all his rewards.

Impact

Player looses all his rewards



Code Snippet

Provided above

Tool used

Manual Review

Recommendation

First of all, you need to allow to call `rebalanceBasket` only once per rebalance period, before new rebalancing period started and allocations are sent to `xController`. Then you need to change check inside `addToTotalRewards` to this `if (currentRebalancingPeriod < lastRebalancingPeriod) return;` in order to allow accruing for same period.



Issue M-40: MainVault.rebalanceXChain doesn't check that savedTotalUnderlying >= reservedFunds

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/107>

Found by

rvierdiiev

Summary

MainVault.rebalanceXChain doesn't check that savedTotalUnderlying >= reservedAmount. Because of that, shortage can occur, if vault will lose some underlying during cross chain calls and reservedFunds will not be present in the vault.

Vulnerability Detail

reservedFunds is the amount that is reserved to be withdrawn by users. It's increased by totalWithdrawalRequests amount every cycle, when setXChainAllocation is called.

setXChainAllocation call is initiated by xController. This call provides vault with information about funds. In case if vault should send funds to the xController, then SendingFundsXChain state is set, aslo amount to send is stored.

After that someone should call MainVault.rebalanceXChain in order to send that amount to the xController. <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L307-L326>

```
function rebalanceXChain(uint256 _slippage, uint256 _relayerFee) external
↳ payable {
    require(state == State.SendingFundsXChain, stateError);

    if (amountToSendXChain > getVaultBalance()) pullFunds(amountToSendXChain);
    if (amountToSendXChain > getVaultBalance()) amountToSendXChain =
↳ getVaultBalance();

    vaultCurrency.safeIncreaseAllowance(xProvider, amountToSendXChain);
    IXProvider(xProvider).xTransferToController{value: msg.value}(
        vaultNumber,
        amountToSendXChain,
        address(vaultCurrency),
        _slippage,
```



```

        _relayerFee
    );

    emit RebalanceXChain(vaultNumber, amountToSendXChain, address(vaultCurrency));

    amountToSendXChain = 0;
    settleReservedFunds();
}

```

As you can see, function just pulls needed funds from providers if needed and sends them to xController. It doesn't check that after that amount that is held by vault is enough to cover reservedFunds. Because of that next situation can occur.

1. Suppose that vault has 1000 tokens as underlying amount. 2. reservedFunds is 200. 3. xController calculated that vault should send 800 tokens to xController (vault allocations is 0) and 200 should be still in the vault in order to cover reservedFunds. 4. when vault is going to send 800 tokens (between setXChainAllocation and rebalanceXChain call), then loss happens and totalUnderlying becomes 800, so currently vault has only 800 tokens in total. 5. vault sends this 800 tokens to xController and has 0 to cover reservedFunds, but actually he should leave this 200 tokens in the vault in this case.

```

if (amountToSendXChain > getVaultBalance()) pullFunds(amountToSendXChain);
if (amountToSendXChain > getVaultBalance()) amountToSendXChain =
↳ getVaultBalance();

```

I think that this is incorrect approach for withdrawing of funds as there is a risk that smth will happen with underlying amount in the providers, so it will be not enough to cover reservedFunds and calculations will be broken, users will not be able to withdraw. Same approach is done in rebalance function, which pulls reservedFunds after depositing to all providers. I guess that correct approach is not to touch reservedFunds amount. In case if you need to send amount to xController, then you need to withdraw it directly from provider. Of course if you have getVaultBalance that is bigger than reservedFunds + amountToSendXChain, then you can send them directly, without pulling.

Impact

Reserved funds protection can be broken

Code Snippet

Provided above



Tool used

Manual Review

Recommendation

You need to check that after you send funds to xController it's enough funds to `cover reservedFunds`.



Issue M-41: Vault.rebalance calculates rewards per locked token incorrectly

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/105>

Found by

rvierdiiev

Summary

Vault.rebalance calculates rewards per locked token incorrectly, because it calls `settleDeltaAllocation` function, which updates `totalAllocatedTokens` variable with new allocations, before `storePriceAndRewards` function, which calculates rewards based on `totalAllocatedTokens` amount.

Vulnerability Detail

Function `storePriceAndRewards` is responsible for calculating rewards per locked token for provider. To calculate rewards it uses `totalAllocatedTokens` variable. This variable should be allocated amount for previous period.

Function `storePriceAndRewards` is called inside `rebalanceCheckProtocols` function, which is called inside `rebalance` function. The problem is that before `rebalanceCheckProtocols` function, the `settleDeltaAllocation` function is called and this function is actually updates `totalAllocatedTokens` with new allocations.

This is the problem as `storePriceAndRewards` function should calculate rewards based on `totalAllocatedTokens` amount for previous period, but because `settleDeltaAllocation` is called before, `totalAllocatedTokens` amount corresponds to the new period and calculations are incorrect.

Impact

Rewards for all reward cycles are incorrect.

Code Snippet

Provided above

Tool used

Manual Review



Recommendation

You need to calculate rewards before changing `totalAllocatedTokens` variable.



Issue M-42: Blacklisting a protocol leads to lower allocations

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/94>

Found by

Met

Summary

Blacklisting a protocol does not update (decrease) the totalAllocatedTokens. It is used to calculate the new allocations and if it is larger than it should be it will lead to lower token allocations to protocols and lower yield.

Vulnerability Detail

Blacklisting does not touch the variable (it decreases the currentAllocations[_protocolNum] but not the total)

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L477-L483>

totalAllocatedTokens is only modified by delta allocations, there is no fix to it.

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L167-L170>

The protocol allocation is calculated hereby, decreased by totalAllocatedTokens value. <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/Vault.sol#L209-L218>

Impact

Forever decreased token allocations to protocols and lower yield.

Code Snippet

Tool used

Manual Review

Recommendation

totalAllocatedTokens by the zeroed currentAllocations[_protocolNum]



Discussion

sjoerdsommen

duplicate with <https://github.com/sherlock-audit/2023-01-derby-judging/issues/95>
<https://github.com/sherlock-audit/2023-01-derby-judging/issues/94>



Issue M-43: Incorrect chainId comparison in xProvider

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/77>

Found by

Sulpiride

Summary

`xProvider.pushTotalUnderlying` and `xProvider.pushRewardsToGame` incorrectly implement `chainId` comparison.

Vulnerability Detail

`xProvider.pushTotalUnderlying` and `xProvider.pushRewardsToGame` incorrectly implement `chainId` comparison. Instead of comparing the `chainId` of `xProvider` with the `chainId` of the destination chain, the `chainId` from functions' parameter was used

Impact

Calling these two functions will fail in two scenarios:

- 1) When `chainId != xControllerChain` (or `gameChain`) and `xProvider.homeChain == xControllerChain`
- 2) When `chainId == xControllerChain` and `xControllerChain != xProvider.homeChain`
 - `chainId = incoming _chainId` from functions parameters

In the first case, the call will fail because you can't xcall to the same chain. In the second case, because `xController` and `game` are `address(0)`

This will prevent some vaults from moving to the next stage, so this just halts them

Code Snippet

`pushTotalUnderlying`: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L207-L245>

```
function pushTotalUnderlying(
    uint256 _vaultNumber,
    uint32 _chainId,
    uint256 _underlying,
    uint256 _totalSupply,
    uint256 _withdrawalRequests
```



```

) external payable onlyVaults {
    if (_chainId == xControllerChain) { // _chainId can be different from
    ↪ homeChain, but equal to xControllerChain
        return
        IXChainController(xController).setTotalUnderlying(
            _vaultNumber,
            _chainId,
            _underlying,
            _totalSupply,
            _withdrawalRequests
        );
    } else {
        bytes4 selector = bytes4(
            keccak256("receiveTotalUnderlying(uint256,uint32,uint256,uint256,uint256)")
        );
        bytes memory callData = abi.encodeWithSelector(
            selector,
            _vaultNumber,
            _chainId,
            _underlying,
            _totalSupply,
            _withdrawalRequests
        );

        xSend(xControllerChain, callData, 0);
    }
}

```

pushRewardsToGame: <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L427-L445>

```

function pushRewardsToGame(
    uint256 _vaultNumber,
    uint32 _chainId,
    int256[] memory _rewards
) external payable onlyVaults {
    if (_chainId == gameChain) {
        return IGame(game).settleRewards(_vaultNumber, _chainId, _rewards);
    } else {
        bytes4 selector =
    ↪ bytes4(keccak256("receiveRewardsToGame(uint256,uint32,int256[])"));
        bytes memory callData = abi.encodeWithSelector(selector, _vaultNumber,
    ↪ _chainId, _rewards);

        xSend(gameChain, callData, 0);
    }
}

```



Tool used

Manual Review

Recommendation

Change the condition to the following:

```
function pushTotalUnderlying(  
  ...  
  -if (_chainId == xControllerChain) {  
+if (homeChain == xControllerChain) {  
  ...
```

```
function pushRewardsToGame(  
  ...  
  -if (_chainId == gameChain) {  
+if (homeChain == gameChain) {  
  ...
```

Discussion

sjoerdsommen

Correct should be (homechain == xChainControllerChain). But medium issue cause no funds are at risk.



Issue M-44: Unprotected slippage tolerance can lead to user/protocol loss of funds

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/54>

Found by

Met, c7e7eff, Ch_301, tives, ff, cergyk, wzrdk3lly

Summary

Slippage is the difference between the expected price of an order and the price when the order actually executes. If the slippage tolerance is set too low, a transaction will not execute. If the slippage tolerance is set too high, users will lose money for paying more per token than intended during swaps. When a rebalance is ready to take place on Derby, any user can front run the `rebalanceXChain()` call setting the slippage tolerance to be 100% for the xChain swap.

Vulnerability Detail

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L307-L320>

1. Any external user/attacker invokes the `rebalanceXChain()` by frontrunning the transaction when it hits the mempool

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L321-L333>

2. When the `homechain` is not equal to the `controllerChain` `xTransfer()` is called with the `_slippage` tolerance

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L152-L159>

3. This XCall will cause the rebalance to be performed, making the proper XChain swaps with the unprotected slippage tolerance.

Impact

Fulfilled swaps where the actual slippage is significantly higher than the standard tolerance of 30 (.03%) would result in loss of funds.

Code Snippet

The entrypoint for this attack occurs in the `rebalanceXChain()` function below <https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser>



/contracts/MainVault.sol#L307-L320

Tool used

Manual Review

Recommendation

Require minimum and maximum values that the protocol is willing to accept for a slippage tolerance that will not cause significant fund loss during Connex XChain swaps. Alternatively the `rebalancXChain()` call can leverage `onlyDao` or `onlyKeeper` modifiers to ensure that these are the only two trusted entities allowed to make this call as intended per the Derby Documentation.



Issue M-45: Did not Approve to zero first

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/36>

Found by

Ruhum, tsvetanovv, R-Nemes, hyh, martin, HonorLt, chainNue, koxuan, Saeedalipoor01988, SunSec

Summary

Vulnerability Detail

Some ERC20 tokens like USDT require resetting the approval to 0 first before being able to reset it to another value. The ohm.approve, pairToken.approve, pool.approve function does not do this - unlike OpenZeppelin's safeApprove implementation.

Impact

unsafe ERC20 approve that do not handle non-standard erc20 behavior. 1. Some token contracts do not return any value. 2. Some token contracts revert the transaction when the allowance is not zero.

Code Snippet

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L150>

```
// This contract approves transfer to Connex  
IERC20(_token).approve(address(connex), _amount);
```

Tool used

Manual Review

Recommendation

It is recommended to set the allowance to zero before increasing the allowance and use safeApprove/safeIncreaseAllowance.



Issue M-46: maxTrainingDeposit can be bypassed

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/15>

Found by

chaduke, csanuragjain, c7e7eff

Summary

It was observed that User can bypass the maxTrainingDeposit by transferring balance from one user to another

Vulnerability Detail

1. Observe the deposit function

```
function deposit(
    uint256 _amount,
    address _receiver
) external nonReentrant onlyWhenVaultIsOn returns (uint256 shares) {
    if (training) {
        require(whitelist[msg.sender]);
        uint256 balanceSender = (balanceOf(msg.sender) * exchangeRate) / (10 **
↪ decimals());
        require(_amount + balanceSender <= maxTrainingDeposit);
    }
    ...
}
```

2. So if User balance exceeds maxTrainingDeposit then request fails (considering training is true)
3. Lets say User A has balance of 50 and maxTrainingDeposit is 100
4. If User A deposit amount 51 then it fails since $50+51 \leq 100$ is false
5. So User A transfer amount 50 to his another account
6. Now when User A deposit, it does not fail since $0+51 \leq 100$

Impact

User can bypass maxTrainingDeposit and deposit more than allowed



Code Snippet

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/MainVault.sol#L113>

Tool used

Manual Review

Recommendation

If user specific limit is required then transfer should be check below:

```
require(_amountTransferred + balanceReceipient <= maxTrainingDeposit);
```

Discussion

sjoerdsommen

yes this is known, there is no solution to this and this is also not a huge problem



Issue M-47: ERC20 return values not checked

Source: <https://github.com/sherlock-audit/2023-01-derby-judging/issues/10>

Found by

Met, Saeedalipoor01988, tsvetanovv, oot2k, R-Nemes, Bnke0x0, PRAISE, hyh, gmx, 8olidity, SunSec, Avci

Summary

Vulnerability Detail

Some tokens (like USDT) don't correctly implement the EIP20 standard and their transfer/transferFrom function return void instead of a successful boolean. Calling these functions with the correct EIP20 function signatures will always revert.

Impact

Tokens that don't correctly implement the latest EIP20 spec, like USDT, will be unusable in the protocol as they revert the transaction because of the missing return value. As there is a xToken with USDT as the underlying issue directly applies to the protocol.

Code Snippet

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L147>

```
IERC20(_token).transferFrom(msg.sender, address(this), _amount);
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L329>

```
IERC20(_asset).transferFrom(msg.sender, xController, _amount);
```

<https://github.com/sherlock-audit/2023-01-derby/blob/main/derby-yield-optimiser/contracts/XProvider.sol#L372>

```
IERC20(_asset).transferFrom(msg.sender, _vault, _amount);
```

Tool used

Manual Review



Recommendation

We recommend using OpenZeppelin's SafeERC20 versions with the `safeTransfer` and `safeTransferFrom` functions that handle the return value check as well as non-standard-compliant tokens.

