SHERLOCK

Security Review For Mento

Public contest prepared for: **Mento**
Lead Security Expert: **0x73696d616f**
Date Audited: **October 24 - October 31, 2024**

Introduction

This upgrade adds a GoodDollar-specific exchange provider that implements the Bancor formula and allows GoodDollar expansions from reserve rewards. Effectively, it allows the GoodDollar team to relaunch the GoodDollar UBI token on top of Mento's asset issuance and redemption engine with an isolated reserve that can collect protocol rewards and expand the supply of GoodDollar.

Scope

Repository: mento-protocol/mento-core

Branch: develop

Audited Commit: 8722c6da3bb8a161996ca0b8fc2a4d0847b0916c

Final Commit: 20fc515c055dcf44f68c0bbbb3dec223be6bea2a

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

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

Issues found

High	Medium
0	5

Issues not fixed or acknowledged

High	Medium
0	0

Security experts who found valid issues

0x73696d616f
Robert
Ollam
onthehunt
0xc0ffEE

Issue M-1: User to sell the last supply will make the exchange contribution forever stuck

Source: https://github.com/sherlock-audit/2024-10-mento-update-judging/issues/17

The protocol has acknowledged this issue.

Found by

0x73696d616f

Summary

BancorExchangeProvider::_getScaledAmountOut() decreases the amount out when the token in is the supply token by the exit contribution, that is, `scaledAmountOut=(scaledAmountOut*(MAX_WEIGHT-exchange.exitContribution))/MAX_WEIGHT;`.

Whenever the last supply of the token is withdrawn, it will get all the reserve and store it in `scaledAmountOut`, and then apply the exit contribution, leaving these funds forever stuck, as there is 0 supply to redeem it.

Root Cause

In `BancorExchangeProvider:345`, the exchange contribution is applied regardless of there being supply left to redeem it.

Internal pre-conditions

1. All supply must be withdrawn from the exchange.

External pre-conditions

None.

Attack Path

1. Users call `Broker::swapIn()`, that calls `GoodDollarExchangeProvider::swapIn()`, which is the exchange contract that holds the token and reserve balances, selling supply tokens until the supply becomes 0.

Impact

The last exit contribution will be forever stuck. This amount is unbounded and may be very significant.

PoC

Add the following test to `BancorExchangeProvider.t.sol`:

```
function test_POC_swapIn_whenTokenInIsToken_shouldSwapIn() public {
  BancorExchangeProvider bancorExchangeProvider =
↪  initializeBancorExchangeProvider();
  uint256 amountIn =  300_000 * 1e18;

  bytes32 exchangeId = bancorExchangeProvider.createExchange(poolExchange1);

  vm.startPrank(brokerAddress);
  uint256 amountOut = bancorExchangeProvider.swapIn(exchangeId, address(token),
↪  address(reserveToken), amountIn);

  (, , uint256 tokenSupplyAfter, uint256 reserveBalanceAfter, , ) =
↪  bancorExchangeProvider.exchanges(exchangeId);

  assertEq(amountOut, 59400e18);
  assertEq(reserveBalanceAfter, 600e18);
  assertEq(tokenSupplyAfter, 0);

  vm.expectRevert("ERR_INVALID_SUPPLY");
  bancorExchangeProvider.swapIn(exchangeId, address(token), address(reserveToken),
↪  1e18);
}
```

Mitigation

The specific mitigation depends on the design.

Issue M-2: `GoodDollarExchangeProvider::mintFromExpansion()` will change the price due to a rounding error in the new ratio

Source: https://github.com/sherlock-audit/2024-10-mento-update-judging/issues/21

Found by

0x73696d616f

Summary

GoodDollarExchangeProvider::mintFromExpansion() mints supply tokens while keeping the current price constant. To achieve this, a certain formula is used, but in the process it scales the `reserveRatioScalar*exchange.reserveRatio` to `1e8` precision (the precision of `exchange.reserveRatio`) down from `1e18`.

However, the calculation of the new amount of tokens to mint is based on the full ratio with 1e18, which will mint more tokens than it should and change the price, breaking the readme.

Note1: there is also a slight price change in GoodDollarExchangeProvider::mintFromInterest() due to using `mul` and then `div`, as `mul` divides by `1e18` unnecessarily in this case.

Note2: GoodDollarExchangeProvider::updateRatioForReward() also has precision loss as it calculates the ratio using the formula and then scales it down, changing the price.

Root Cause

In `GoodDollarExchangeProvider:147`, `newRatio` is calculated with full `1e18` precision and used to calculate the amount of tokens to mint, but `exchanges[exchangeId].reserveRatio` is stored with the downscaled value, `newRatio/1e10`, causing an error and price change.

This happens because the price is `reserve/(supply*reserveRatio)`. As `supply` is increased by a calculation that uses the full precision `newRatio`, but `reserveRatio` is stored with less precision (`1e8`), the price will change due to this call.

Internal pre-conditions

None.

External pre-conditions

None.

Attack Path

1. `GoodDollarExchangeProvider::mintFromExpansion()` is called and a rounding error happens in the calculation of `newRatio`.

Impact

The current price is modified due to the expansion which goes against the readme:

> What properties/invariants do you want to hold even if breaking them has a low/unknown impact?
>
> Bancor formula invariant. Price = Reserve / Supply * reserveRatio

PoC

Add the following test to `GoodDollarExchangeProvider.t.sol`:

```
function test_POC_mintFromExpansion_priceChangeFix() public {
  uint256 priceBefore = exchangeProvider.currentPrice(exchangeId);
  vm.prank(expansionControllerAddress);
  exchangeProvider.mintFromExpansion(exchangeId, reserveRatioScalar);
  uint256 priceAfter = exchangeProvider.currentPrice(exchangeId);
  assertEq(priceBefore, priceAfter, "Price should remain exactly equal");
}
```

If the code is used as is, it fails. but if it is fixed by dividing and multiplying by `1e10`, eliminating the rounding error, the price matches exactly (exact fix show below).

Mitigation

Divide and multiply `newRatio` by `1e10` to eliminate the rounding error, keeping the price unchanged.

```
function mintFromExpansion(
  bytes32 exchangeId,
  uint256 reserveRatioScalar
) external onlyExpansionController whenNotPaused returns (uint256 amountToMint) {
  require(reserveRatioScalar > 0, "Reserve ratio scalar must be greater than 0");
  PoolExchange memory exchange = getPoolExchange(exchangeId);

  UD60x18 scaledRatio = wrap(uint256(exchange.reserveRatio) * 1e10);
```

```
  UD60x18 newRatio = wrap(unwrap(scaledRatio.mul(wrap(reserveRatioScalar))) / 1e10
↪  * 1e10);
  ...
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/mento-protocol/mento-core/pull/548

Issue M-3: Malicious user may frontrun `GoodDollarExpansionController::mintUBIFromReserveBalance()` to make protocol funds stuck

Source: https://github.com/sherlock-audit/2024-10-mento-update-judging/issues/33

The protocol has acknowledged this issue.

Found by

0x73696d616f

Summary

GoodDollarExpansionController::mintUBIFromReserveBalance() or GoodDollarExpansionController::mintUBIFromInterest() transfer funds to the reserve and mint $G to the distribution helper. However, GoodDollarExchangeProvider::mintFromInterest() mints 0 tokens whenever the supply is 0. An attacker can buy all $G from the exchange to trigger this.

Root Cause

In `GoodDollarExpansionController::142` and `GoodDollarExpansionController::161`, `amountMinted` is not checked for a null value.

Internal pre-conditions

None.

External pre-conditions

None.

Attack Path

1. Attacker calls `Bancor::swapIn()` or `Bancor::swapOut()`, buying all $G in the exchange, making `PoolExchange.tokenSupply` null.

2. `GoodDollarExpansionController::mintUBIFromReserveBalance()` or `GoodDollarExpansionController::mintUBIFromInterest()` is called, adding reserve asset funds without minting $G.

Impact

Funds are added to the reserve without the corresponding amount of $G being minted.

PoC

`GoodDollarExpansionController::mintUBIFromInterest()` and `GoodDollarExpansionController::mintUBIFromReserveBalance()` do not check if `amountToMint` is null:

```
function mintUBIFromInterest(bytes32 exchangeId, uint256 reserveInterest) external {
  require(reserveInterest > 0, "Reserve interest must be greater than 0");
  IBancorExchangeProvider.PoolExchange memory exchange =
↪   IBancorExchangeProvider(address(goodDollarExchangeProvider))
    .getPoolExchange(exchangeId);

  uint256 amountToMint = goodDollarExchangeProvider.mintFromInterest(exchangeId,
↪   reserveInterest);

  require(IERC20(exchange.reserveAsset).transferFrom(msg.sender, reserve,
↪   reserveInterest), "Transfer failed"); //@audit safeTransferFrom.  //@audit lost
↪   if reserve asset is also a stable asset
  IGoodDollar(exchange.tokenAddress).mint(address(distributionHelper),
↪   amountToMint);

  // Ignored, because contracts only interacts with trusted contracts and tokens
  // slither-disable-next-line reentrancy-events
  emit InterestUBIMinted(exchangeId, amountToMint);
}

...

function mintUBIFromReserveBalance(bytes32 exchangeId) external returns (uint256
↪   amountMinted) {
  IBancorExchangeProvider.PoolExchange memory exchange =
↪   IBancorExchangeProvider(address(goodDollarExchangeProvider))
    .getPoolExchange(exchangeId);

  uint256 contractReserveBalance = IERC20(exchange.reserveAsset).balanceOf(reserve);
  uint256 additionalReserveBalance = contractReserveBalance -
↪   exchange.reserveBalance;
  if (additionalReserveBalance > 0) {
    amountMinted = goodDollarExchangeProvider.mintFromInterest(exchangeId,
↪   additionalReserveBalance);
```

```
    IGoodDollar(exchange.tokenAddress).mint(address(distributionHelper),
↪   amountMinted);

    // Ignored, because contracts only interacts with trusted contracts and tokens
    // slither-disable-next-line reentrancy-events
    emit InterestUBIMinted(exchangeId, amountMinted);
  }
}
```

`GoodDollarExchangeProvider::mintFromInterest()` returns 0 if `exchange.tokenSupply` is 0.

```
function mintFromInterest(
  bytes32 exchangeId,
  uint256 reserveInterest
) external onlyExpansionController whenNotPaused returns (uint256 amountToMint) {
  PoolExchange memory exchange = getPoolExchange(exchangeId);

  uint256 reserveinterestScaled = reserveInterest *
↪   tokenPrecisionMultipliers[exchange.reserveAsset];
  uint256 amountToMintScaled = unwrap(
    wrap(reserveinterestScaled).mul(wrap(exchange.tokenSupply)).div(wrap(exchange.r
↪   eserveBalance))
  );
  amountToMint = amountToMintScaled /
↪   tokenPrecisionMultipliers[exchange.tokenAddress];

  exchanges[exchangeId].tokenSupply += amountToMintScaled;
  exchanges[exchangeId].reserveBalance += reserveinterestScaled;

  return amountToMint;
}
```

Mitigation

Revert if the `amountToMint` from the `GoodDollarExchangeProvider::mintFromInterest()` call is null. The same should also be done for `GoodDollarExpansionController::mintUBIFromExpansion()` `amountMinted` from the `GoodDollarExchangeProvider.mintFromExpansion()` call.

Issue M-4: `TradingLimits::update()` incorrectly only rounds up when `deltaFlowUnits` becomes 0, which will silently increase trading limits

Source: https://github.com/sherlock-audit/2024-10-mento-update-judging/issues/45

The protocol has acknowledged this issue.

Found by

0x73696d616f

Summary

TradingLimits::update() divides the traded funds by the decimals of the token, int256 _deltaFlowUnits = _deltaFlow / int256((10 ** uint256(decimals))); . `Inatokenwith18decimals`, `forexample,swapping1.999...e18tokenswillleadtoa`_deltaFlowUnits`ofjust1`‘, taking a major error. This can be exploited to swap up to twice the trading limit, if tokens are swapped 2 by 2 and the state is updated only by 1 each time. Overall, even without malicious intent, the limits will always be bypassed due to the rounding.

Root Cause

In `TradingLimits:135`, it only rounds up whenever `deltaFlowUnits` becomes 0, but the error is just as big if it becomes 1 from 2, effectively not providing enough protection.

Internal pre-conditions

None.

External pre-conditions

None.

Attack Path

1. User calls `Broker::swapIn/Out()` with amounts in and out that produce rounding errors (almost always).

Impact

The trading limits may be severely bypassed with malicious intent (by double the amount) or by a smaller but still significant amount organically.

PoC

`TradingLimits::update()` only rounds up when `deltaFlowUnits` becomes 0.

```
function update(
  ITradingLimits.State memory self,
  ITradingLimits.Config memory config,
  int256 _deltaFlow,
  uint8 decimals
) internal view returns (ITradingLimits.State memory) {
  int256 _deltaFlowUnits = _deltaFlow / int256((10 ** uint256(decimals)));
  require(_deltaFlowUnits <= MAX_INT48, "dFlow too large");

  int48 deltaFlowUnits = int48(_deltaFlowUnits);
  if (deltaFlowUnits == 0) {
    deltaFlowUnits = _deltaFlow > 0 ? int48(1) : int48(-1);
  }
  ...
```

Mitigation

The correct fix is:

```
int256 _deltaFlowUnits = (_deltaFlow - 1) / int256((10 ** uint256(decimals))) + 1;
```

Issue M-5: _getReserveRatioScalar() will give a lesser value than expected

Source: https://github.com/sherlock-audit/2024-10-mento-update-judging/issues/50

Found by

0x73696d616f, 0xc0ffEE, Ollam, Robert, onthehunt

Summary

numberOfExpansions = (block.timestamp - config.lastExpansion) / config.expansionFrequency

The calculation divides it by the expansionFrequency, but this will cause significant rounding issues.

If the expansionFrequency is 1 day (as specified in the docs), time may pass without anybody calling the function and the following scenario will be present.

Let's say 30 hours since last expansion and someone decides then to call it, it will be rounded due to the division to be 1 day, producing a smaller value than the hours that've passed.

Root Cause

The root cause is the potential of **rounding down** `numberOfExpansions`, which will give a significantly smaller value, depending on how big will be remainder of the division. (6 for 30 hours, 3 for 27 hours, etc)

Internal pre-conditions

`mintUBIFromExpansion()` need to be callable.

External pre-conditions

No response

Attack Path

1. Alice calls `mintUBIFromExpansion()` to create an expansion

2. 30 hours pass and nobody calls the function, Bob sees that he can call `mintUBIFromExpansion()`
3. Due to the rounding down of the calculation, it will a value equivalent of 24 hours passing.

Impact

The protocol will expand **slower than intended**, thus less $G will be minted, which will **become significant** overtime.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/mento-protocol/mento-core/pull/553

Disclaimers

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

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