AstraSec

DeltaPrime Tokenomics

Security Audit Report

August 7, 2024

Contents

1 Introduction

1.1 About DeltaPrime Tokenomics

DeltaPrime tokenomics introduces three tokens ($PRIME, $sPRIME, $vPRIME) to deepen pool liquidity as much as possible. $PRIME can be traded on the open market, or combined with another asset to create $sPRIME. $sPRIME is a utility token, used as the main currency to pay for any Prime Brokerage service within the DeltaPrime ecosystem. $vPRIME is DeltaPrime's governance token, which builds up over time by staking $sPRIME combined with protocol exposure on deposit/borrow-side.

1.2 Source Code

The audit scope covers the code changes in below PR:

- https://github.com/DeltaPrimeLabs/deltaprime-primeloans/pull/304/
- CommitID: f9fc6b5

And this is the final repository and commit hash after all fixes for the issues found in the audit have been checked in:

- https://github.com/DeltaPrimeLabs/deltaprime-primeloans/pull/304/
- CommitID: ef4bdca

Note that this audit only covers the smart contracts in the `contracts/token/` folder, including `Prime.sol`, `PrimeBridge.sol`, `Prime_L2.sol`, `PositionManager.sol`, `sPrime.sol`, `sPrimeUniswap.sol`, `vPrime.sol` and `vPrimeController.sol`, etc.

2 | Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the tokens contracts of `DeltaPrime`. Throughout this audit, we identified a total of 6 issues spanning various severity levels. By employing auxiliary tool techniques to supplement our thorough manual code review, we have discovered the following findings.

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	-	-	-	-
High	2	-	-	2
Medium	4	2	-	2
Low	-	-	-	-
Informational	-	-	-	-
Total	6	2	-	4

3 Vulnerability Summary

3.1 Overview

Click on an issue to jump to it, or scroll down to see them all.

~~H-1~~ Revised Liquidity Removal Logic in sPrimeUniswap

~~H-2~~ Incorrect Token Conversion Formula in sPrime

M-1 Manipulated Position Value in sPrimeUniswap/sPrime

~~M-2~~ Incorrect sPrime Value Calculation in vPrimeController

~~M-3~~ Wrong Token Approval in sPrimeUniswap::_swapForEqualValues()

M-4 Potential Risks Associated with Centralization

3.2 Security Level Reference

In web3 smart contract audits, vulnerabilities are typically classified into different severity levels based on the potential impact they can have on the security and functionality of the contract. Here are the definitions for critical-severity, high-severity, medium-severity, and low-severity vulnerabilities:

Severity	Description
C-X (Critical)	A severe security flaw with immediate and significant negative consequences. It poses high risks, such as unauthorized access, financial losses, or complete disruption of functionality. Requires immediate attention and remediation.
H-X (High)	Significant security issues that can lead to substantial risks. Although not as severe as critical vulnerabilities, they can still result in unauthorized access, manipulation of contract state, or financial losses. Prompt remediation is necessary.
M-X (Medium)	Moderately impactful security weaknesses that require attention and remediation. They may lead to limited unauthorized access, minor financial losses, or potential disruptions to functionality.
L-X (Low)	Minor security issues with limited impact. While they may not pose significant risks, it is still recommended to address them to maintain a robust and secure smart contract.
I-X (Informational)	Warnings and things to keep in mind when operating the protocol. No immediate action required.
U-X (Undetermined)	Identified security flaw requiring further investigation. Severity and impact need to be determined. Additional assessment and analysis are necessary.

3.3 Vulnerability Details

[H-1] Revised Liquidity Removal Logic in sPrimeUniswap

Target	Category	IMPACT	LIKELIHOOD	STATUS
sPrimeUniswap.sol	Business Logic	High	High	Addressed

In the `DeltaPrime` protocol, the `sPrimeUniswap` contract facilitates users' deposits and withdrawals in `UniswapV3`. It mints or burns `sPrime` tokens for users based on the amounts of tokens deposited or withdrawn. Specifically, in `UniswapV3`, the assets withdrawn from removing liquidity are not automatically transferred to the users. Instead, users need to manually call the `collect()` function to withdraw their assets. While examining the implementation of the `sPrimeUniswap` contract, we notice that it does not properly collect the withdrawn assets from `UniswapV3` in the `deposit()`, `withdraw()`, and `_afterTokenTransfer()` functions.

To elaborate, we show below the code snippet of `sPrimeUniswap::deposit()`, which is used to rebalance the existing position for the user. Specifically, it removes all liquidity from the position (line 317), burns all the user's `sPrime` tokens (line 327), burns the position NFT from the `positionManager` (line 328), deletes the recorded token ID for the removed position (line 330), and updates the token amounts for the subsequent deposit (line 332). However, it does not collect the withdrawn assets from `UniswapV3` after removing liquidity. As a result, it will fail to burn the position from the `positionManager`, causing the transaction to revert.

Note that the same issue also exists in the `withdraw()` and `_afterTokenTransfer()` functions.

sPrimeUniswap::deposit()

```
316 if(isRebalance) { // Withdraw Position For Rebalance
317   (uint256 amountXBefore, uint256 amountYBefore) = INonfungiblePositionManager(
          positionManager).decreaseLiquidity(
318       INonfungiblePositionManager.DecreaseLiquidityParams({
319           tokenId: tokenId,
320           liquidity: liquidity,
321           amount0Min:0,
322           amount1Min:0,
323           deadline: block.timestamp
324       })
325   );

327   _burn(_msgSender(), balanceOf(_msgSender()));
328   INonfungiblePositionManager(positionManager).burn(tokenId);

330   delete userTokenId[_msgSender()];

332   (amountX, amountY) = (amountX + amountXBefore, amountY + amountYBefore);
```

```
333 }
```

What is more, if a user withdraws all the liquidity from `UniswapV3` via the `withdraw()` function, it calls `positionManager.burn(tokenId)` to burn the position NFT from `UniswapV3` (line 377). However, it does not delete the recorded `userTokenId[msgSender]` for the burnt position NFT. As a result, it records a legacy token ID for the burned position in the contract.

sPrimeUniswap::withdraw()

```
365   (uint256 amountX, uint256 amountY) = INonfungiblePositionManager(
          positionManager).decreaseLiquidity(
366       INonfungiblePositionManager.DecreaseLiquidityParams({
367           tokenId: tokenId,
368           liquidity: uint128(liquidity * share / balanceOf(_msgSender())),
369           amount0Min:0,
370           amount1Min:0,
371           deadline: block.timestamp
372       })
373 );

375 // Burn Position NFT
376 if(balanceOf(_msgSender()) == share) {
377     INonfungiblePositionManager(positionManager).burn(tokenId);
378 }

380 _burn(_msgSender(), share);

382 // Send the tokens to the user.
383 _transferTokens(address(this), _msgSender(), amountX, amountY);
```

Remediation Properly collect assets from liquidity withdrawal in the `deposit()`, `withdraw()` and `_afterTokenTransfer()` functions, and delete the `userTokenId[msgSender]` for the burnt position.

[H-2] Incorrect Token Conversion Formula in sPrime

Target	Category	IMPACT	LIKELIHOOD	STATUS
sPrime.sol	Coding Practices	High	High	Addressed

In the `DeltaPrime` protocol, the `sPrime` contract is used to interact with the `TraderJoe` protocol and mint `sPrime` tokens based on the amount of tokens deposited into `TraderJoe`. During our review of the deposit processing, we identified an issue with the incorrect calculation of token conversion between tokenX and tokenY.

To elaborate, we show below the related code snippet of the `_getTokenYFromTokenX()` function, which is used to convert the given amount of tokenX to tokenY based on the active pool price. Specifically, it calls the `PriceHelper.convert128x128PriceToDecimal()` function to read the price of tokenX in tokenY(line 214), and calculates the amount of tokenY using the formula $amountY = amountX / price$ (line 216). As the price represents the value of tokenX, the correct formula should be $amountY = amountX * price$.

_getTokenYFromTokenX()

```
211 function _getTokenYFromTokenX(uint256 amountX) internal view returns(uint256
        amountY) {
212     (uint128 reserveA, ) = lbPair.getReserves();
213     if(reserveA > 0) {
214         uint256 price = PriceHelper.convert128x128PriceToDecimal(lbPair.
                getPriceFromId(lbPair.getActiveId()));
215         // Swap For Y : Convert token X to token Y
216         amountY = amountX * (10 ** IERC20Metadata(address(tokenY)).decimals()) /
                 price;
217     } else {...}
218 }
```

Similar issue exists in the `_swapForEqualValues()` function, where the conversion from tokenY to tokenX should use the formula $amountX = amountY / price$ instead of $amountX = amountY * price$ (line 242).

_swapForEqualValues()

```
227     /**
228     * @dev Returns the updated amounts of tokens.
229     * @return amountX The updated amount of token X.
230     * @return amountY The updated amount of token Y.
231     */
232 function _swapForEqualValues(uint256 amountX, uint256 amountY, uint256
        swapSlippage) internal returns(uint256, uint256) {
233     uint256 amountXToY = _getTokenYFromTokenX(amountX);
234     bool swapTokenX = amountY < amountXToY;
235     uint256 diff = swapTokenX ? amountXToY - amountY : amountY - amountXToY;
236     // (amountXToY != 0  amountX == 0) for excluding the initial LP deposit
237     if(amountY * _REBALANCE_MARGIN / 100 < diff && (amountXToY > 0  amountX ==
            0)) {
238         uint256 amountIn;
239         {
240             uint256 price = PriceHelper.convert128x128PriceToDecimal(lbPair.
                    getPriceFromId(lbPair.getActiveId()));
241             // Swap For X : Convert token Y to token X
242             amountIn = (diff / 2) * price / (10 ** IERC20Metadata(address(tokenY
                    )).decimals());
243         }
```

```
            ...
        }
        ...
}
```

Remediation Use the correct formula for the token conversion between tokenX and tokenY.

[M-1] Manipulated Position Value in sPrimeUniswap/sPrime

Target	Category	IMPACT	LIKELIHOOD	STATUS
sPrimeUniswap.sol, sPrime.sol	Business Logic	Medium	Medium	Acknowledged

In the `sPrimeUniswap` contract, the `getUserValueInTokenY()` function is used to calculate the position value for the given user. The position value is used by `vPrimeController` to distribute the governance token `vPrime`. While examining the calculation of the position value, we notice the possibility that the position value can be manipulated.

In the following, we show the code snippet of the `sPrimeUniswap::getUserValueInTokenY()` function. Specifically, it reads the active price from the pool by calling `pool.slot0()` (line 127), gets the total amounts of tokenX and tokenY for the given position by calling `positionManager.total(tokenId, sqrtRatioX96)` (line 130), and calculates the position value in tokenY by calling `_getTotalInTokenY()` (line 131).

However, it comes to our attention that it uses the active pool price to calculate the total amounts of tokenX and tokenY for the position in the same pool. Specifically, the active pool price can be inflated by a front-run swap, leading to the position value being inflated. Our study shows that it should use a robust price oracle, such as `Chainlink` or `RedStone`, as the price feed to calculate the position value.

sPrimeUniswap::getUserValueInTokenY()

```
123 function getUserValueInTokenY(address user) public view returns (uint256) {
124     uint256 tokenId = userTokenId[user];
125     require(tokenId > 0, "No position");

127     (uint160 sqrtRatioX96,,,,,,) = pool.slot0();

129     address positionManager = getNonfungiblePositionManagerAddress();
130     (uint256 amountX, uint256 amountY) = INonfungiblePositionManager(
            positionManager).total(tokenId, sqrtRatioX96);
131     return _getTotalInTokenY(amountX, amountY);
132 }
```

Note that the same issue also exists in the `sPrime::_getTokenYFromTokenX()` function. As shown in the code snippet below, it reads the price of the active bin from the `lbPair` (line 214) , and uses the price to calculate the amount of tokenY for the given amount of tokenX (line 216). Because the active bin in the `lbPair` can be moved left or right by a swap, the calculated amount of tokenY can be manipulated.

sPrime::_getTokenYFromTokenX()

```
211 function _getTokenYFromTokenX(uint256 amountX) internal view returns(uint256
        amountY) {
212     (uint128 reserveA, ) = lbPair.getReserves();
213     if(reserveA > 0) {
214         uint256 price = PriceHelper.convert128x128PriceToDecimal(lbPair.
                getPriceFromId(lbPair.getActiveId()));
215         // Swap For Y : Convert token X to token Y
216         amountY = amountX * (10 ** IERC20Metadata(address(tokenY)).decimals()) /
                 price;
217     } else {
218         amountY = 0;
219     }
220 }
```

Remediation Choose a robust price oracle, such as `Chainlink` or `RedStone`, as the price feed to calculate the position value.

Response By Team This issue has been acknowledged by the team: We'll be using the price from Oracle couple of days after TGE (once the price feed is available) and in the first days we'll be using the pool-reported price but with a max cap on the `PRIME` price being 10x the listing price. As this only influences `vPrime` accrual, limiting this to 10x and not having an oracle only for couple of days (out of 3 years max `vPrime` accrual) is an acceptable risk for us.

[M-2] Incorrect sPrime Value Calculation in vPrimeController

Target	Category	IMPACT	LIKELIHOOD	STATUS
vPrimeController.sol	Business Logic	Medium	Medium	Addressed

In the `DeltaPrime` protocol, the `vPrimeController` contract is responsible for updating the vPrime snapshot for users. As one of the key factors in calculating the `vPrime` distribution amount, it gets users' position values deposited via the `sPrime` contracts. While reviewing the implementation of the `getUserSPrimeDollarValueVestedAndNonVested()` function, we notice that wrong formulas are applied.

In the following, we show the related code snippet from the `vPrimeController` contract. Specifically, it gets the position value in tokenY for the given user in each `sPrime` (line 157), and calculates the `fullyVestedDollarValue` and the `nonVestedDollarValue` respectively according to their balance proportions in the `sPrime` (lines $159-160$). However, it comes to our attention that it does not correctly calculate the balance proportions for `fullyVestedBalance` and `nonVestedBalance`. For the balance proportion of `fullyVestedBalance`, it mistakenly uses the formula $fullyVestedDollarValue/sPrimeBalance$ instead of the correct formula $fullyVestedBalance/sPrimeBalance$. Similarly, for the balance proportion of `nonVestedBalance`, it mistakenly uses the formula $nonVestedDollarValue/sPrimeBalance$ instead of the correct formula $nonVestedBalance/sPrimeBalance$.

What is more, it should add the new non-vested dollar value to `nonVestedDollarValue`, but it mistakenly add it to `nonVestedBalance` (line 160).

vPrimeController::getUserSPrimeDollarValueVestedAndNonVested()

```
147 function getUserSPrimeDollarValueVestedAndNonVested(address userAddress) public
        view returns (uint256 fullyVestedDollarValue, uint256 nonVestedDollarValue) {
148     fullyVestedDollarValue = 0;
149     nonVestedDollarValue = 0;
150     for (uint i = 0; i < whitelistedSPrimeContracts.length; i++) {
151         bytes32 sPrimeTokenYSymbol = tokenManager.tokenAddressToSymbol(
                whitelistedSPrimeContracts[i].getTokenY());
152         uint256 sPrimeTokenYDecimals = IERC20Metadata(whitelistedSPrimeContracts[
                i].getTokenY()).decimals();
153         uint256 sPrimeTokenYPrice = getOracleNumericValueFromTxMsg(
                sPrimeTokenYSymbol);
154         uint256 sPrimeBalance = whitelistedSPrimeContracts[i].balanceOf(
                userAddress);
155         uint256 fullyVestedBalance = whitelistedSPrimeContracts[i].
                getFullyVestedLockedBalance(userAddress);
156         uint256 nonVestedBalance = sPrimeBalance - fullyVestedBalance;
157         uint256 userSPrimeValueInTokenY = whitelistedSPrimeContracts[i].
                getUserValueInTokenY(userAddress);

159         fullyVestedDollarValue += userSPrimeValueInTokenY * sPrimeTokenYPrice * 1
                e10 * fullyVestedDollarValue / sPrimeBalance / 10 **
                sPrimeTokenYDecimals;
160         nonVestedBalance += userSPrimeValueInTokenY * sPrimeTokenYPrice * 1e10 *
                nonVestedDollarValue / sPrimeBalance / 10 ** sPrimeTokenYDecimals;
161     }
162     return (fullyVestedDollarValue, nonVestedDollarValue);
163 }
```

Remediation Correct the formulas used to calculate the position values in each `sPrime`.

[M-3] Wrong Token Approval in sPrimeUniswap::_swapForEqualValues()

Target	Category	IMPACT	LIKELIHOOD	STATUS
sPrimeUniswap.sol	Coding Practices	Medium	Medium	Addressed

In the `sPrimeUniswap` contract, the internal `_swapForEqualValues()` function is used to equalize the values of tokenX and tokenY for the input token amounts. This is achieved by swapping some of one token for the other through interacting with the `swapRouter` of `Uniswap`. Before interacting with the `swapRouter` to perform the swap, it needs to grant the `swapRouter` permission to spend the specified amount of the input token.

As shown in the code below, when the input token to swap is tokenX, it grants the `swapRouter` permission to spend tokenX (line 211). However, when the input token is tokenY, it still grants the `swapRouter` permission to spend tokenX (line 215), instead of the expected tokenY. As a result, the subsequent swap will fail due to a lack of allowance for tokenY.

sPrimeUniswap::_swapForEqualValues()

```
198 function _swapForEqualValues(uint256 amountX, uint256 amountY, uint256 swapSlippage) internal returns(uint256, uint256) {
199   ...
200   uint256 amountOut = diff / 2;

202   address swapRouter = getSwapRouter();
203   address tokenIn;
204   address tokenOut;

206   (amountIn, amountOut) = swapTokenX ? (amountIn, amountOut) : (amountOut, amountIn);

208   if (swapTokenX) {
209       tokenIn = address(tokenX);
210       tokenOut = address(tokenY);
211       tokenX.safeApprove(swapRouter, amountIn);
212   } else {
213       tokenIn = address(tokenY);
214       tokenOut = address(tokenX);
215       tokenX.safeApprove(swapRouter, amountIn);
216   }
217   ...
218 }
```

Remediation Properly grant the `swapRouter` permission to spend tokenY when the input token is tokenY.

[M-4] Potential Risks Associated with Centralization

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple contracts	Security	High	Low	Acknowledged

In the `DeltaPrime` protocol, the existence of a privileged `owner` account introduces centralization risks, as it holds significant control and authority over critical operations governing the protocol. In the following, we show the representative functions potentially affected by the privileges associated with the privileged `owner` account.

Example Privileged Operations in `vPrimeController`

```
function updateWhitelistedSPrimeContracts(SPrimeMock[] memory
    newWhitelistedSPrimeContracts) external onlyOwner {
  whitelistedSPrimeContracts = newWhitelistedSPrimeContracts;
  emit WhitelistedSPrimeContractsUpdated(newWhitelistedSPrimeContracts, msg.
      sender, block.timestamp);
}

/**
* @notice Updates the token manager contract.
* @dev Can only be called by the contract owner.
* @param newTokenManager The address of the new token manager contract.
*/
function updateTokenManager(ITokenManager newTokenManager) external onlyOwner {
  tokenManager = newTokenManager;
  emit TokenManagerUpdated(newTokenManager, msg.sender, block.timestamp);
}

/**
* @notice Updates the borrowers registry contract.
* @dev Can only be called by the contract owner.
* @param newBorrowersRegistry The address of the new borrowers registry contract.
*/
function updateBorrowersRegistry(IBorrowersRegistry newBorrowersRegistry)
    external onlyOwner {
  borrowersRegistry = newBorrowersRegistry;
}
```

Example Privileged Operations in `vPrime`

```
function setVPrimeControllerAddress(address _vPrimeControllerAddress) external
    onlyOwner {
  vPrimeControllerAddress = _vPrimeControllerAddress;
}
```

Remediation To mitigate the issue, it is recommended to introduce multi-sig mechanism to undertake the role of the privileged account. Moreover, it is advisable to implement timelocks to govern all modifications to the privileged operations.

Response By Team This issue has been mitigated by implementing a multisig and timelock mechanism to manage the owner.

4 | Conclusion

The `DeltaPrime tokenomics` protocol introduces three tokens (`$PRIME`, `$sPRIME`, `$vPRIME`) to deepen pool liquidity. It increases the power of users' usual DeFi investment, and contributes to the network's security and stability. The current code base is well-structured and neatly organized. Those identified issues have been promptly acknowledged and fixed.

5 Appendix

5.1 About AstraSec

AstraSec is a blockchain security company that serves to provide high-quality auditing services for blockchain-based protocols. With a team of blockchain specialists, AstraSec maintains a strong commitment to excellence and client satisfaction. The audit team members have extensive audit experience for various famous DeFi projects. AstraSec's comprehensive approach and deep blockchain understanding make it a trusted partner for the clients.

5.2 Disclaimer

The information provided in this audit report is for reference only and does not constitute any legal, financial, or investment advice. Any views, suggestions, or conclusions in the audit report are based on the limited information and conditions obtained during the audit process and may be subject to unknown risks and uncertainties. While we make every effort to ensure the accuracy and completeness of the audit report, we are not responsible for any errors or omissions in the report.

We recommend users to carefully consider the information in the audit report based on their own independent judgment and professional advice before making any decisions. We are not responsible for the consequences of the use of the audit report, including but not limited to any losses or damages resulting from reliance on the audit report.

This audit report is for reference only and should not be considered a substitute for legal documents or contracts.

5.3 Contact

Phone	+86 176 2267 4194
Email	contact@astrasec.ai
Twitter	https://twitter.com/AstraSecAI