BLOCKSEC

Security Audit

Report for DeltaPrime Contracts

Date: September 22, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	DeltaPrime
Target	DeltaPrime Contracts

Version History

Version	Date	Description
1.0	September 22, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-10-14 02:41:52+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

DeltaPrime is a decentralized lending platform on Avalanche and Arbitrum. Depositors can lend funds by depositing an asset into one of the liquidity pools, earning interest in return. Borrowers can perform under-collateralized borrowing from these liquidity pools. Specifically, DeltaPrime introduces a special-purpose smart contract (i.e., the prime account) to enable borrowers to access liquidity pools and invest their borrowed funds across various DeFi protocols integrated by DeltaPrime. For security, DeltaPrime implements a series of checks to ensure the solvency of prime accounts throughout the borrowing and investing process. Borrowers are only allowed to withdraw their funds once all debts have been repaid. Additionally, a group of liquidators is introduced to mitigate insolvency risk by partially repaying unhealthy loans.

This audit focuses on DeltaPrime Contracts [1] of DeltaPrime. Specifically, the files covered in this audit are listed below.

List of files included:

- contracts/*

List of files excluded:

- contracts/abstract/ECDSAVerify.sol
- contracts/abstract/NFTAccess.sol
- contracts/helpers/*
- contracts/integrations/celo/*
- contracts/interfaces/*
- contracts/lib/joe-v2/*
- contracts/lib/celo/*
- contracts/mock/*
- contracts/token/mock/*
- contracts/VestingDistributor.sol
- contracts/facets/celo/*
- contracts/facets/mock/*
- contracts/facets/RecoveryFacet.sol
- contracts/facets/SmartLoanLiquidationFacetDebug.sol
- contracts/facets/arbitrum/ConvexFacetArbi.sol
- contracts/facets/arbitrum/RemoveWstEthAssetFacet.sol
- contracts/facets/avalanche/CurveFacet.sol

[1] https://github.com/DeltaPrimeLabs/deltaprime-primeloans

- contracts/facets/avalanche/GMDFacet.sol
- contracts/facets/avalanche/VectorFinanceFacetOld.sol

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the versions `Version 1-1` and `Version 1-2` [2], as well as new code (in the following versions) to fix issues in the audit report. The commits for `Version 1-1` and `Version 1-2` are not present in the repository due to repository management actions taken by the project during the audit process.

Project	Version	Commit Hash
DeltaPrime Contracts	`Version 1-1`	`9179e9244dfa38cdf3d78e015402c583390e0cb7`
	`Version 1-2`	`24598f83404821ff69711243a33d5bfb3b027e5e` [3]
	`Version 2`	`f3b956ac5992c0bc144af5a106aba2783b589109`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.

[2] This version introcudes a withdrawal intent feature during the audit process.

[3] This commit was included in `PR#362`.

- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [4] and Common Weakness Enumeration [5]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[4] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[5] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **thirty-nine** potential security issues. Besides, we have **twenty-one** recommendations and **eleven** notes.

- High Risk: 8
- Medium Risk: 9
- Low Risk: 22
- Recommendation: 21
- Note: 11

ID	Severity	Description	Category	Status
1	High	Improper slot assignment in the `_unsafeAccess` function	Software Security	Fixed
2	High	Incorrect `vPrime` snapshot update due to manipulable vote balance	DeFi Security	Fixed
3	High	Double deduction in the `_afterTokenTransfer` function	DeFi Security	Fixed
4	High	Manipulable `vPrimeRate` due to dependency on spot price	DeFi Security	Fixed
5	High	Improper share calculation due to spot price dependency	DeFi Security	Fixed
6	High	Unexpected liquidation due to malicious unfreezing	DeFi Security	Fixed
7	High	Flawed duplication check for bin IDs causing position value inflation	DeFi Security	Fixed
8	High	Unexpected claim of borrowed assets due to the lack of validation for pairs	DeFi Security	Fixed
9	Medium	Manipulable protocol exposure due to the lack of access control	DeFi Security	Fixed
10	Medium	Potential precision loss in the `getPoolPrice` function	DeFi Security	Fixed
11	Medium	Incorrect decimal handling in the `getUserValueInTokenY` function	DeFi Security	Fixed
12	Medium	Unexpected adjustment to the received `sPrime` amount due to spot price dependency	DeFi Security	Fixed
13	Medium	Incorrect return value in the `getMaximumTokensReceived` function	DeFi Security	Fixed
14	Medium	Improper implementations violating the withdrawal guard design	DeFi Security	Fixed
15	Medium	Unclaimable fee in the `UniswapV3Facet` contract	DeFi Security	Fixed

16	Medium	Unenforced modifier `noBorrowInTheSameBlock` in specific functions	DeFi Security	Fixed
17	Medium	Flawed design allowing stealing of assets from insolvent prime accounts	DeFi Security	Fixed
18	Low	Potential bypass of whitelisted borrower limitation	DeFi Security	Fixed
19	Low	Potential DoS due to the lack of pop operation for `locks`	DeFi Security	Fixed
20	Low	Potential DoS due to incomplete removal of unsupported assets	DeFi Security	Fixed
21	Low	Unfair reward distribution due to inconsistent deposit records	DeFi Security	Confirmed
22	Low	Lack of check on the asset distinction when swapping debts	DeFi Security	Fixed
23	Low	Incorrect return value due to unsorted `depositForm`	DeFi Security	Fixed
24	Low	Unhandled native asset in the `claimReward` function	DeFi Security	Fixed
25	Low	Double-counting issue due to the lack of duplication check for the variable `whitelistedSPrimeContracts`	DeFi Security	Fixed
26	Low	Temporary DoS in the `fundLiquidityTraderJoeV2` function	DeFi Security	Fixed
27	Low	Potential excessive deduction of protocol exposure	DeFi Security	Fixed
28	Low	Unexpected operations by liquidator	DeFi Security	Fixed
29	Low	Unexpected revert due to dust token in the `liquidate` function	DeFi Security	Fixed
30	Low	Inability to track real-time exposure	DeFi Security	Fixed
31	Low	Inconsistent health factor calculation between contracts `SolvencyFacetProd` and `HealthMeterFacetProd`	DeFi Security	Fixed
32	Low	Improper modifier usage in the `swapDebtParaSwap` function	DeFi Security	Fixed
33	Low	Lack of validation for duplicate users in `eligibleUsersList`	DeFi Security	Fixed
34	Low	Lack of a reentrancy guard in the `createAndFundLoan` function	DeFi Security	Fixed
35	Low	Lack of validation for the parameter `_adapters` in the `yakSwap` function	DeFi Security	Fixed

36	Low	Lack of validation for additional rewards received	DeFi Security	Fixed
37	Low	Improper condition check in the `createWithdrawalIntent` function	DeFi Security	Fixed
38	Low	Reusable withdrawal intents due to reentrancy issue	DeFi Security	Fixed
39	Low	Incomplete withdrawal due to exclusion of accrued interest	DeFi Security	Fixed
40	-	Remove the unused `prevIndex` assignment in the `updateIndex` function	Recommendation	Fixed
41	-	Remove the unused state variable `_status` in the `ReentrancyGuardKeccak` contract	Recommendation	Fixed
42	-	Remove the redundant validation in the `unwrapAndWithdraw` function	Recommendation	Fixed
43	-	Remove the redundant import in the `AssetsOperationsAvalancheFacet` contract	Recommendation	Fixed
44	-	Remove the redundant validation in the functions `_withdrawFromPool` and `_depositToPool`	Recommendation	Fixed
45	-	Ensure balance check before removing pool assets	Recommendation	Fixed
46	-	Implement a validation check for the function's parameters	Recommendation	Fixed
47	-	Remove the redundant `payable` modifier	Recommendation	Fixed
48	-	Add a validation check to prevent the executing meaningless transactions	Recommendation	Fixed
49	-	Unify the handling of `msg.value` in the `_getTWVOwnedAssets` function across different chains	Recommendation	Fixed
50	-	Remove redundant `view` functions in the `DiamondStorageLib` contract	Recommendation	Fixed
51	-	Remove the redundant price query	Recommendation	Fixed
52	-	Remove redundant validation checks in the `mintAndStakeGlp` function	Recommendation	Fixed
53	-	Refactor the misleading annotation in the `getLiquidityTokenAmounts` function	Recommendation	Fixed
54	-	Add access control checks to the `refundExecutionFee` function	Recommendation	Fixed

55	-	Allow a customizable `callbackGasLimit` in the `GmxV2Facet` and `GmxV2PlusFacet` contracts	Recommendation	Confirmed
56	-	Add whitelist checks while interacting with external contracts	Recommendation	Fixed
57	-	Revise the reward handling logic in the `TraderJoeV2Facet` contract	Recommendation	Fixed
58	-	Revise the contracts inheriting from the `ReentrancyGuardKeccak` contract	Recommendation	Fixed
59	-	Add validation checks for the `msg.value` and `executionFee`	Recommendation	Fixed
60	-	Implement the redeem logic in the `GogoPoolFacet` contract	Recommendation	Fixed
61	-	Potential centralization risks	Note	-
62	-	Reentrancy risks in pools for tokens with callback mechanisms	Note	-
63	-	Potential integration risks	Note	-
64	-	Potential deployment risks	Note	-
65	-	Malicious transfer of `sPrime` prevents users from receiving shares with their desired `tokenId`	Note	-
66	-	Potential withdrawal failure in pools with rebasing, inflation, or deflation tokens	Note	-
67	-	Deprecated contracts	Note	-
68	-	Configure the variable `delay` properly	Note	-
69	-	Ensure proper synchronization of vPrime tokens	Note	-
70	-	Ensure intentional withdrawal design in liquidation processes	Note	-
71	-	Integrate the RedStone oracle with a proper setting	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Improper slot assignment in the `_unsafeAccess` function

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `vPrime` contract, the function `_unsafeAccess` is used to return a `Checkpoint` from the `ckpts` array (i.e., `Checkpoint[]`) at the index `pos`. The function employs assembly code to calculate the starting slot of the targeted `Checkpoint` and assigns it to `result.slot`. However, the calculation for the starting slot directly uses the index `pos` without considering the size of the `Checkpoint` struct. As a result, `_unsafeAccess` returns an incorrect `result`, which subsequently causes errors in the calculations performed by the `getPastVotes` and `getPastTotalSupply` functions.

```
343   function _unsafeAccess(Checkpoint[] storage ckpts, uint256 pos) private pure returns (
          Checkpoint storage result) {
344       assembly {
345           mstore(0, ckpts.slot)
346           result.slot := add(keccak256(0, 0x20), pos)
347       }
348   }
```

Listing 2.1: contracts/token/vPrime.sol

```
296       uint256 length = ckpts.length;
297
298       uint256 low = 0;
299       uint256 high = length;
300
301       if (length > 5) {
302           uint256 mid = length - Math.sqrt(length);
303           if (_unsafeAccess(ckpts, mid).blockTimestamp > timestamp) {
304               high = mid;
305           } else {
306               low = mid + 1;
307           }
308       }
```

Listing 2.2: contracts/token/vPrime.sol

```
192   function getPastVotes(address account, uint256 timestamp) public view virtual returns (uint256)
          {
193       require(timestamp < clock(), "Future lookup");
194       return _checkpointsLookup(_checkpoints[account], timestamp);
195   }
```

Listing 2.3: contracts/token/vPrime.sol

```
205   function getPastTotalSupply(uint256 timestamp) public view virtual returns (uint256) {
206       require(timestamp < clock(), "Future lookup");
207       return _checkpointsLookup(_checkpoints[address(this)], timestamp);
208   }
```

Listing 2.4: contracts/token/vPrime.sol

Impact The problematic slot assignment in the `_unsafeAccess` function could lead to incorrect return values in the `getPastVotes` and `getPastTotalSupply` functions.

Suggestion Revise the slot assignment logic in the function `_unsafeAccess` by accounting for the size of the struct `Checkpoin`.

2.2 DeFi Security

2.2.1 Incorrect `vPrime` snapshot update due to manipulable vote balance

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `vPrime` contract, the function `balanceOf` calculates the amount of a user's accumulated `vPrime` tokens. Specifically, if the state `needsUpdate[account]` is *true*, it returns the `account`'s last recorded balance (i.e., `cp.balance`). Otherwise it recalculates the balance as `cp.balance +elapsedTime * cp.rate`. Moreover, in the `adjustRateAndCap` function, this balance is used to calculate `voteDiff`, which is then used to create a new checkpoint.

However, malicious users can manipulate the `needsUpdate` state of any account by invoking the `transfer` or `transferFrom` functions in the `pool` contract without including oracle calldata. As a result, the `balanceOf` function returns only `cp.balance` without accounting for the accumulated part (i.e., `elapsedTime * cp.rate`). This design flaw enables malicious users to increase their voting power by setting other accounts' `needsUpdate` state to *true*.

```
113   function balanceOf(address account) public view returns (uint256) {
114       uint256 userCkpsLen = _checkpoints[account].length;
115       if (userCkpsLen == 0) {
116           return 0;
117       }
118       Checkpoint memory cp = _checkpoints[account][userCkpsLen - 1];
119
120       // If account was updated without recalculations, return the last recorded balance
121       if(needsUpdate[account]) {
122           return cp.balance;
123       }
124
125       uint256 elapsedTime = block.timestamp - cp.blockTimestamp;
126       uint256 newBalance;
127
128       if (cp.rate >= 0) {
129           uint256 balanceIncrease = uint256(cp.rate) * elapsedTime;
130           newBalance = cp.balance + balanceIncrease;
131           return (newBalance > cp.balanceLimit) ? cp.balanceLimit : newBalance;
132       } else {
133           // If rate is negative, convert to positive for calculation, then subtract
134           uint256 balanceDecrease = uint256(- cp.rate) * elapsedTime;
135           if (balanceDecrease > cp.balance) {
136               // Prevent underflow, setting balance to min cap if decrease exceeds current balance
137               return cp.balanceLimit;
138           } else {
139               newBalance = cp.balance - balanceDecrease;
140               return (newBalance < cp.balanceLimit) ? cp.balanceLimit : newBalance;
141           }
142       }
143   }
```

Listing 2.5: contracts/token/vPrime.sol

Impact Malicious users can decrease any account's `vPrime` balance, thereby increasing their voting power.

Suggestion Revise the code logic accordingly.

2.2.2 Double deduction in the `_afterTokenTransfer` function

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `sPrimeUniswap` contract, the hook function `_afterTokenTransfer` is invoked to transfer liquidity from the `sPrime` sender (i.e., the input `from`) to the `sPrime` receiver (i.e., the input `to`). Specifically, this function mints a new position for the receiver using the sender's liquidity (i.e., `amountXAdded` and `amountYAdded`). It then transfers the remaining asset (i.e., `amountX -= amountXAdded` and `amountY -= amountYAdded`) to the receiver. However, the `_afterTokenTransfer` function performs this deduction twice (on lines 833-834 and 846-847), resulting in a double deduction of the remaining assets. This logic error could cause a denial of service (DoS) issue for `sPrime` transfers or lead to the receiver receiving an incorrect amount of the remaining assets.

```
785   function _afterTokenTransfer(
786       address from,
787       address to,
788       uint256 amount
789   ) internal virtual override {
790       // ...
791
792               (uint256 amountX, uint256 amountY) = positionManagerRemove(tokenId, uint128((
                      liquidity * amount) / (fromBalance + amount)), address(this), 0, 0);
793               getToken0().forceApprove(address(positionManager), amountX);
794               getToken1().forceApprove(address(positionManager), amountY);
795               (
796                   uint256 newTokenId,
797                   ,
798                   uint256 amountXAdded,
799                   uint256 amountYAdded
800               ) = positionManager.mint(
801                       INonfungiblePositionManager.MintParams({
802                           token0: address(getToken0()),
803                           token1: address(getToken1()),
804                           fee: feeTier,
805                           tickLower: tickLower,
806                           tickUpper: tickUpper,
807                           amount0Desired: amountX,
808                           amount1Desired: amountY,
809                           amount0Min: 0,
810                           amount1Min: 0,
```

```
811                         recipient: address(this),
812                         deadline: block.timestamp
813                     })
814                 );
815             amountX -= amountXAdded;
816             amountY -= amountYAdded;
817
818             if(getToken0() != tokenX) {
819                 (amountXAdded, amountYAdded) = (amountYAdded, amountXAdded);
820                 (amountX, amountY) = (amountY, amountX);
821             }
822
823             uint256 total = _getTotalInTokenY(amountXAdded, amountYAdded);
824
825             _transferTokens(
826                 address(this),
827                 to,
828                 amountX - amountXAdded,
829                 amountY - amountYAdded
830             );
```

Listing 2.6: contracts/token/sPrimeUniswap.sol

Impact Potential DoS on transferring tokens in the contract `sPrimeUniswap`.

Suggestion Revise the code logic accordingly.

2.2.3 Manipulable `vPrimeRate` due to dependency on spot price

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `vPrimeController`, the function `getPrimeTokenPoolPrice` returns a spot price for a UniswapV3-based `sPrime` contract when `useOraclePrimeFeed` is set to *false*. This price is then used to calculate the `userSPrimeValueInTokenY`, which determines the `sPrime` dollar value in the function `getUserSPrimeDollarValueVestedAndNonVested`.

```
199   function getPrimeTokenPoolPrice(ISPrime sPrimeContract, uint256 tokenYPrice) public view
          returns (uint256) {
200       if(useOraclePrimeFeed){
201           bytes32 primeSymbol = "PRIME";
202           uint256 primePrice = getOracleNumericValueFromTxMsg(primeSymbol);
203           return primePrice * 1e8 / tokenYPrice; // both tokenYPrice and primePrice have 8
                  decimals
204       } else {
205           uint256 poolPrice = sPrimeContract.getPoolPrice(); // returns price with 8 decimals
206           if(poolPrice * tokenYPrice / 1e8 > 13125 * 1e5){ // 13.125 with 8 decimals which comes
                  from 10xinitialPrice MAX PRICE CAP before oracle feed will be ready
207               poolPrice = 13125 * 1e13 / tokenYPrice; // 13125 * 1e5 * 1e8 / tokenYPrice
208           }
209           return poolPrice;
210       }
```

```
211   }
```

Listing 2.7: contracts/token/vPrimeController.sol

```
269   function getPoolPrice() public view returns (uint256) {
270       (, int24 tick, , , , , ) = pool.slot0();
271       uint256 price = OracleLibrary.getQuoteAtTick(
272           tick,
273           uint128(10 ** tokenX.decimals()),
274           address(tokenX),
275           address(tokenY)
276       );
277       return
278           FullMath.mulDiv(
279               price,
280               1e8,
281               10 ** tokenY.decimals()
282           );
283   }
```

Listing 2.8: contracts/token/sPrimeUniswap.sol

```
213   function getUserSPrimeDollarValueVestedAndNonVested(address userAddress) public view returns (
          uint256 fullyVestedDollarValue, uint256 nonVestedDollarValue) {
214       fullyVestedDollarValue = 0;
215       nonVestedDollarValue = 0;
216       for (uint i = 0; i < whitelistedSPrimeContracts.length; i++) {
217           // ...
218           uint256 userSPrimeValueInTokenY = whitelistedSPrimeContracts[i].getUserValueInTokenY(
                  userAddress, poolPrice);
219           if(sPrimeBalance > 0) {
220               uint256 _denominator = sPrimeBalance * 10 ** sPrimeTokenYDecimals;
221               fullyVestedDollarValue += FullMath.mulDiv(userSPrimeValueInTokenY, sPrimeTokenYPrice
                      * RS_PRICE_PRECISION_1e18_COMPLEMENT * fullyVestedBalance, _denominator);
222               nonVestedDollarValue += FullMath.mulDiv(userSPrimeValueInTokenY, sPrimeTokenYPrice *
                      RS_PRICE_PRECISION_1e18_COMPLEMENT * nonVestedBalance, _denominator);
223           }
224       }
225       return (fullyVestedDollarValue, nonVestedDollarValue);
226   }
```

Listing 2.9: contracts/token/vPrimeController.sol

However, the spot price used to calculate the `sPrime` dollar value could be easily manipulated via a flash loan, which could affect the calculation of `vPrimeBalanceLimit` and `vPrimeRate` within the function `getUserVPrimeRateAndMaxCap`. Specifically, an attacker could artificially lower the spot price to an extremely low value and then update a victim's `vPrime` snapshot using the `updateVPrimeSnapshot` function with the manipulated pool price. During this update, the variable `vPrimeCalculations.vPrimeBalanceLimit` is impacted and adjusted to a lower value due to the manipulated spot price, further decreasing `vPrimeCalculations.vPrimeRate`. Consequently, the victim's voting power is reduced, requiring active intervention and time to recover.

```
239   function getUserVPrimeRateAndMaxCap(address userAddress) public view returns (int256, uint256,
          uint256){
240       VPrimeCalculationsStruct memory vPrimeCalculations = VPrimeCalculationsStruct({
241           vPrimeRate: 0,
242           vPrimeBalanceLimit: 0,
243           vPrimeBalanceAlreadyVested: 0,
244           userSPrimeDollarValueFullyVested: 0,
245           userSPrimeDollarValueNonVested: 0,
246           userDepositFullyVestedDollarValue: 0,
247           userDepositNonVestedDollarValue: 0,
248           primeAccountBorrowedDollarValue: 0
249       });
250
251       {
252           (uint256 _userSPrimeDollarValueFullyVested, uint256 _userSPrimeDollarValueNonVested) =
                  getUserSPrimeDollarValueVestedAndNonVested(userAddress);
253           (uint256 _userDepositFullyVestedDollarValue, uint256 _userDepositNonVestedDollarValue) =
                   getUserDepositDollarValueAcrossWhiteListedPoolsVestedAndNonVested(userAddress);
254           uint256 _primeAccountBorrowedDollarValue =
                  getPrimeAccountBorrowedDollarValueAcrossWhitelistedPools(userAddress);
255           vPrimeCalculations.userSPrimeDollarValueFullyVested = _userSPrimeDollarValueFullyVested;
256           vPrimeCalculations.userSPrimeDollarValueNonVested = _userSPrimeDollarValueNonVested;
257           vPrimeCalculations.userDepositFullyVestedDollarValue =
                  _userDepositFullyVestedDollarValue;
258           vPrimeCalculations.userDepositNonVestedDollarValue = _userDepositNonVestedDollarValue;
259           vPrimeCalculations.primeAccountBorrowedDollarValue = _primeAccountBorrowedDollarValue;
260       }
261
262
263       // How many pairs can be created based on the sPrime
264       uint256 maxSPrimePairsCount = (vPrimeCalculations.userSPrimeDollarValueFullyVested +
              vPrimeCalculations.userSPrimeDollarValueNonVested) / 1e18;
265       // How many pairs can be created based on the deposits
266       uint256 maxDepositPairsCount = ((vPrimeCalculations.userDepositFullyVestedDollarValue +
              vPrimeCalculations.userDepositNonVestedDollarValue) / V_PRIME_PAIR_RATIO) / 1e18;
267       // How many pairs can be created based on the borrowings
268       uint256 maxBorrowerPairsCount = vPrimeCalculations.primeAccountBorrowedDollarValue /
              V_PRIME_PAIR_RATIO / 1e18;
269
270       // How many sPrime-depositor pairs can be created
271       uint256 maxSPrimeDepositorPairsCount = Math.min(maxSPrimePairsCount, maxDepositPairsCount);
272       // How many sPrime-borrower pairs can be created taken into account sPrime used by sPrime-
              depositor pairs
273       uint256 maxSPrimeBorrowerPairsCount = Math.min(maxSPrimePairsCount -
              maxSPrimeDepositorPairsCount, maxBorrowerPairsCount);
274
275       // Increase vPrimeCalculations.vPrimeBalanceLimit and vPrimeCalculations.
              vPrimeBalanceAlreadyVested based on the sPrime-depositor pairs
276       if(maxSPrimeDepositorPairsCount > 0){
277           uint256 balanceLimitIncrease = maxSPrimeDepositorPairsCount *
                  DEPOSITOR_YEARLY_V_PRIME_RATE * MAX_V_PRIME_VESTING_YEARS * 1e18;
278           vPrimeCalculations.vPrimeBalanceLimit += balanceLimitIncrease;
279
```

```
280             uint256 depositVestedPairsCount = Math.min(vPrimeCalculations.
                    userDepositFullyVestedDollarValue / V_PRIME_PAIR_RATIO, vPrimeCalculations.
                    userSPrimeDollarValueFullyVested) / 1e18;
281             if(depositVestedPairsCount > 0){
282                 vPrimeCalculations.vPrimeBalanceAlreadyVested += balanceLimitIncrease *
                        depositVestedPairsCount / maxSPrimeDepositorPairsCount;
283             }
284         }
285
286         // Increase vPrimeCalculations.vPrimeBalanceLimit based on the sPrime-borrower pairs
287         if(maxSPrimeBorrowerPairsCount > 0){
288             vPrimeCalculations.vPrimeBalanceLimit += maxSPrimeBorrowerPairsCount *
                    BORROWER_YEARLY_V_PRIME_RATE * MAX_V_PRIME_VESTING_YEARS * 1e18;
289         }
290
291         // Check current vPrime balance
292         uint256 currentVPrimeBalance = vPrimeContract.balanceOf(userAddress);
293
294         // If already vested vPrime balance is higher than the current balance, then the current
                balance should be replaced with the already vested balance
295         bool balanceShouldBeReplaced = false;
296         if(currentVPrimeBalance < vPrimeCalculations.vPrimeBalanceAlreadyVested){
297             currentVPrimeBalance = vPrimeCalculations.vPrimeBalanceAlreadyVested;
298             balanceShouldBeReplaced = true;
299         }
300
301         int256 vPrimeBalanceDelta = int256(vPrimeCalculations.vPrimeBalanceLimit) - int256(
                currentVPrimeBalance);
302         if(vPrimeBalanceDelta < 0){
303             vPrimeCalculations.vPrimeRate = vPrimeBalanceDelta / int256(V_PRIME_DETERIORATION_DAYS)
                    / 1 days;
304         } else {
305             vPrimeCalculations.vPrimeRate = vPrimeBalanceDelta / int256(MAX_V_PRIME_VESTING_YEARS) /
                     365 days;
306         }
307
308         if(balanceShouldBeReplaced){
309             return (vPrimeCalculations.vPrimeRate, vPrimeCalculations.vPrimeBalanceLimit,
                    vPrimeCalculations.vPrimeBalanceAlreadyVested);
310         } else {
311             return (vPrimeCalculations.vPrimeRate, vPrimeCalculations.vPrimeBalanceLimit, 0);
312         }
313     }
```

Listing 2.10: contracts/token/vPrimeController.sol

Impact Users could experience a decrease in voting power due to spot price manipulation.

Suggestion Avoid using the spot price for value calculation.

Feedback from the project The project stated that the price oracle is properly integrated (i.e., the flag `useOraclePrimeFeed` is set to true) for price feeding, and the fixed price (i.e., `13125 * 1e13 / tokenYPrice`) is not in use now.

2.2.4 Improper share calculation due to spot price dependency

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `sPrimeUniswap`, the function `deposit` is used to add liquidity to the UniswapV3 pool and mint corresponding shares (i.e., `sPrime` token). Specifically, in the function `_depositToUniswap`, the amount of `sPrime` tokens minted is based on the total amount of `tokenY`, which includes the amount of `tokenY` converted from `tokenX` using the pool's spot price. However, the spot price could be easily manipulated via a flash loan, leading to an increase in the amount of `sPrime` tokens minted. For example, an attacker with an existing position could first manipulate the pool's spot price (i.e., decrease the price of the `tokenY`) through a flash loan. Then, the attacker can invoke the function `deposit` with customized inputs (e.g., `isRebalance` is set to `false`) to ensure that the liquidity is only added to the existing position. In the function `_depositToUniswap`, since the manipulated spot price is used to convert the `tokenX` to `tokenY`, the attacker can receive a larger amount of `sPrime` tokens. Consequently, the attacker could exploit this vulnerability to increase their voting power when updating their `vPrime` snapshot.

```
368   function _depositToUniswap(
369       address user,
370       int24 tickLower,
371       int24 tickUpper,
372       uint256 amountX,
373       uint256 amountY,
374       uint256 desiredAmountX,
375       uint256 desiredAmountY
376   ) internal {
377       uint256 tokenId = userTokenId[user];
378       uint256 amountXAdded;
379       uint256 amountYAdded;
380       tokenX.forceApprove(address(positionManager), amountX);
381       tokenY.forceApprove(address(positionManager), amountY);
382
383       (uint256 amount0, uint256 amount1) = tokenSequence ? (amountY, amountX) : (amountX, amountY
              );
384       (desiredAmountX, desiredAmountY) = tokenSequence ? (desiredAmountY, desiredAmountX) : (
              desiredAmountX, desiredAmountY);
385
386       if (tokenId == 0) {
387           (tokenId, , amountXAdded, amountYAdded) = positionManager.mint(
388               INonfungiblePositionManager.MintParams({
389                   token0: address(getToken0()),
390                   token1: address(getToken1()),
391                   fee: feeTier,
392                   tickLower: tickLower,
393                   tickUpper: tickUpper,
394                   amount0Desired: amount0,
395                   amount1Desired: amount1,
396                   amount0Min: desiredAmountX,
397                   amount1Min: desiredAmountY,
```

```
398                 recipient: address(this),
399                 deadline: block.timestamp
400             })
401         );
402         userTokenId[user] = tokenId;
403     } else {
404         (, amountXAdded, amountYAdded) = positionManager.increaseLiquidity(
405             INonfungiblePositionManager.IncreaseLiquidityParams({
406                 tokenId: tokenId,
407                 amount0Desired: amount0,
408                 amount1Desired: amount1,
409                 amount0Min: desiredAmountX,
410                 amount1Min: desiredAmountY,
411                 deadline: block.timestamp
412             })
413         );
414     }
415     if(tokenSequence) {
416         (amountXAdded, amountYAdded) = (amountYAdded, amountXAdded);
417     }
418
419     uint256 share = _getTotalInTokenY(amountXAdded, amountYAdded);
420
421     _mint(user, share);
```

Listing 2.11: contracts/token/sPrimeUniswap.sol

```
244 function _getTotalInTokenY(
245     uint256 amountX,
246     uint256 amountY
247 ) internal view returns (uint256 weight) {
248     uint256 amountXToY = _getTokenYFromTokenX(amountX);
249     weight = amountY + amountXToY;
250 }
```

Listing 2.12: contracts/token/sPrimeUniswap.so

```
257 function _getTokenYFromTokenX(
258     uint256 amountX
259 ) internal view returns (uint256 amountY) {
260     (, int24 tick, , , , , ) = pool.slot0();
261     amountY = OracleLibrary.getQuoteAtTick(
262         tick,
263         uint128(amountX),
264         address(tokenX),
265         address(tokenY)
266     );
267 }
```

Listing 2.13: contracts/token/sPrimeUniswap.so

Impact The attacker could receive more `sPrime` tokens through a spot price manipulation, leading to an increase in voting power.

Suggestion Revise the code logic accordingly.

2.2.5 Unexpected liquidation due to malicious unfreezing

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `GmxV2Facet` contract, the `_deposit` function allows users to create a deposit order on the GmxV2 protocol. Due to the asynchronous design of the GmxV2 protocol (i.e., the delay in deposit execution), the `_deposit` function invokes the `freezeAccount` function to freeze the prime account before the receipt of `gmToken`, thereby preventing accidental liquidation. Once the deposit order (initiated from the DeltaPrime side) is processed by the GmxV2 protocol, the `gmToken` is sent to the prime account and the callback function `afterDepositExecution` (in the `GmxV2CallbacksFacet` contract) is triggered to unfreeze the prime account.

```
19  function _deposit(
20      address gmToken,
21      address depositedToken,
22      uint256 tokenAmount,
23      uint256 minGmAmount,
24      uint256 executionFee
25  ) internal nonReentrant noBorrowInTheSameBlock onlyOwner {
26      // ...
27
28      // Freeze account
29      DiamondStorageLib.freezeAccount(gmToken);
```

Listing 2.14: contracts/facets/GmxV2Facet.sol

```
59  function afterDepositExecution(bytes32 key, Deposit.Props memory deposit, EventUtils.
        EventLogData memory eventData) external onlyGmxV2Keeper nonReentrant override {
60      // ...
61
62      // Unfreeze account
63      DiamondStorageLib.unfreezeAccount(msg.sender);
```

Listing 2.15: contracts/facets/GmxV2CallbacksFacet.sol

However, the design is problematic due to transaction order dependency. Since the unfreezing operation is triggered in the callback function during the execution of the deposit order, an attacker can potentially front-run this callback to unfreeze the prime account before it receives the `gmToken`, causing the prime account to face unexpected liquidation.

To better understand the potential attack, we provide a detailed example:

1. At block 100, a user creates a deposit order on the GmxV2 protocol via their prime account. Assume the user's position is insolvent, excluding the `gmToken`.
2. At block 101, an attacker creates a deposit order on the GmxV2 protocol, setting the `callbackContract` to the victim's prime account. Assume that the attacker's deposit order is executed before the user's deposit order.
3. At block 102, the attacker's deposit order is executed before the victim's. The callback function `afterDepositExecution` (in the victim's prime account) is triggered, unfreezing

the victim's prime account. Since the prime account's position is insolvent without receiving the `gmToken`, the prime account may suffer liquidation.

Impact Users could face unexpected liquidations due to malicious unfreezing.

Suggestion Revise the code logic accordingly.

2.2.6 Flawed duplication check for bin IDs causing position value inflation

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `TraderJoeV2Facet`, the functions `fundLiquidityTraderJoeV2` and `addLiquidityTraderJoeV2` allow users to add liquidity to the TraderJoeV2 protocol based on provided bin IDs (i.e., the input `ids`). If the prime account does not already have a bin ID, it will be added to the state variable `ownedTjV2Bins` for position value calculation. However, the validation logic for duplicate bin IDs is flawed, allowing duplicate IDs to be recorded in the state variable `ownedTjV2Bins`. As a result, users' position value can be inflated, increasing their borrowing capability and enabling them to drain all pools.

Below are two vulnerabilities that allow an attacker to insert duplicate IDs into the state variable `ownedTjV2Bins`.

1. `uint` **truncation:** In the contract `TraderJoeV2Facet`, the `fundLiquidityTraderJoeV2` function accepts the input `ids` with the type *uint256*, which is truncated to the type *uint24* before being pushed to the state variable `ownedTjV2Bins`. This design can be exploited to bypass the existence check for bin IDs and insert duplicate IDs into the state variable `ownedTjV2Bins`.
   Specifically, if an attacker already has a position with *id = 1* for their prime account, they can exploit the function `fundLiquidityTraderJoeV2` by providing `ids` as *[uint24.max + 1, uint24.max + 1, ...]* and `amounts` as *[0, 0, ...]*. As a result, the variable `userHadBin` will always be *false*, and the duplicate ID (i.e., *id = 1*, which is derived from *uint24(uint24.max + 1)*) will be inserted into the state variable `ownedTjV2Bins`.
2. **Memory variable for validation:** In the function `fundLiquidityTraderJoeV2` of the contract `TraderJoeV2Facet`, there is no check for duplicate IDs in the parameter `ids`. Furthermore, the memory variable `ownedBins`, which is loaded from the `getOwnedTraderJoeV2Bins` function, is not updated when new IDs are pushed into the state variable `ownedTjV2Bins`. This design flaw allows user-specified `ids` to bypass the duplication check.
   Specifically, assume an attacker does not have bin ID 1. The attacker can exploit the function `fundLiquidityTraderJoeV2` by providing inputs `ids` like *[1, 1, 1, ...]* and `amounts` as *[100, 0, 0, ...]*. Since the memory variable `ownedBins` is not updated during the validation, all duplicate IDs will be pushed into the state variable `ownedTjV2Bins`. The same vulnerability exists in the function `addLiquidityTraderJoeV2`.

```
112   function fundLiquidityTraderJoeV2(ILBPair pair, uint256[] memory ids, uint256[] memory amounts)
          external nonReentrant {
113       if (!isPairWhitelisted(address(pair))) revert TraderJoeV2PoolNotWhitelisted();
114
```

```
115       pair.batchTransferFrom(msg.sender, address(this), ids, amounts);
116
117       TraderJoeV2Bin[] memory ownedBins = getOwnedTraderJoeV2Bins();
118
119       for (uint256 i; i < ids.length; ++i) {
120           bool userHadBin;
121
122           for (int256 j; uint(j) < ownedBins.length; ++j) {
123               if (address(ownedBins[uint(j)].pair) == address(pair)
124                   && ownedBins[uint(j)].id == ids[i]
125               ) {
126                   userHadBin = true;
127                   break;
128               }
129           }
130
131           if (!userHadBin) {
132               getOwnedTraderJoeV2BinsStorage().push(TraderJoeV2Bin(pair, uint24(ids[i])));
133           }
134       }
135
136       if (maxBinsPerPrimeAccount() > 0 && getOwnedTraderJoeV2BinsStorage().length >
              maxBinsPerPrimeAccount()) revert TooManyBins();
137
138       emit FundedLiquidityTraderJoeV2(msg.sender, address(pair), ids, amounts, block.timestamp);
139   }
```

Listing 2.16: contracts/facets/TraderJoeV2Facet.sol

Impact Attackers can insert duplicate bin IDs into the state variable `ownedTjV2Bins`, inflating their position value and enabling them to drain all pools.

Suggestion Revise the code logic accordingly.

2.2.7 Unexpected claim of borrowed assets due to the lack of validation for pairs

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `TraderJoeV2Facet`, the function `claimReward` allows users to claim and transfer rewards directly to `msg.sender`. This function calculates the reward amount (i.e., the `rewardToken` balance difference) by recording the prime account's `rewardToken` balance before and after the invocation of `LBHooksBaseRewarder(baseRewarder).claim(address(this), ids)` (the `.claim` function hereafter for brevity). However, there is a lack of proper validation for the input `pair`, which allows a reentrancy attack using a customized contract `pair`.

With the malicious contract `pair`, an attacker could manipulate the `rewardToken` as the wrapped native token and trigger a reentrancy call (in the `.claim` function) to the `wrapNativeToken` function, creating a balance difference. Since there is no solvency check in the `claimReward` function and no reentrancy protection in the `wrapNativeToken` function, the attacker could first

borrow wrapped native tokens (then convert them into native tokens) using a flash loan and siphon all borrowed assets via the above method. Finally, all borrowed assets will be wrapped and transferred to `msg.sender` at the end of the `claimReward` function.

```
91  function claimReward(ILBPair pair, uint256[] calldata ids) external nonReentrant onlyOwner {
92      ILBHookLens lbHookLens = ILBHookLens(getJoeV2LBHookLens());
93      ILBHookLens.Parameters memory hookLens = lbHookLens.getHooks(address(pair));
94      address baseRewarder = hookLens.hooks;
95
96      if(baseRewarder == address(0)) revert TraderJoeV2NoRewardHook();
97
98      address rewardToken = address(ILBHooksBaseRewarder(baseRewarder).getRewardToken());
99      bool isNative = (rewardToken == address(0));
100     uint256 beforeBalance = isNative ? address(this).balance : IERC20(rewardToken).balanceOf(
            address(this));
101     ILBHooksBaseRewarder(baseRewarder).claim(address(this), ids);
102     uint256 reward = isNative ? address(this).balance - beforeBalance : IERC20(rewardToken).
            balanceOf(address(this)) - beforeBalance;
103     if(reward > 0) {
104         if(isNative) {
105             payable(msg.sender).safeTransferETH(reward);
106         } else {
107             rewardToken.safeTransfer(msg.sender, reward);
108         }
109     }
110 }
```

Listing 2.17: contracts/facets/TraderJoeV2Facet.sol

```
53  function getHooks(address pair) public view returns (Hooks.Parameters memory) {
54      if (msg.sender == address(this)) {
55          bytes32 hooksParameters = ILBPair(pair).getLBHooksParameters();
56
57          return Hooks.decode(hooksParameters);
58      } else {
59          try this.getHooks(pair) returns (Hooks.Parameters memory hooksParameters) {
60              return hooksParameters;
61          } catch {
62              return
63                  Hooks.Parameters(address(0), false, false, false, false, false, false, false,
                        false, false, false);
64          }
65      }
66  }
```

Listing 2.18: LBHooksLens.sol on Arbitrum chain

```
16  function wrapNativeToken(uint256 amount) onlyOwnerOrInsolvent public {
17      require(amount <= address(this).balance, "Not enough native token to wrap");
18      require(amount > 0, "Cannot wrap 0 tokens");
19      IWrappedNativeToken wrapped = IWrappedNativeToken(DeploymentConstants.getNativeToken());
20      wrapped.deposit{value : amount}();
21
```

```
22      ITokenManager tokenManager = DeploymentConstants.getTokenManager();
23      _increaseExposure(tokenManager, address(wrapped), amount);
24
25      emit WrapNative(msg.sender, amount, block.timestamp);
26  }
```

Listing 2.19: contracts/facets/SmartLoanWrappedNativeTokenFacet.sol

Impact The attacker can siphon borrowed assets while in an insolvent state.

Suggestion Revise the code logic accordingly.

2.2.8 Manipulable protocol exposure due to the lack of access control

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `AssetsExposureController` contract, the `resetPrimeAccountAssetsExposure` and `setPrimeAccountAssetsExposure` functions are used to adjust the protocol's exposure based on the prime account's assets. However, both functions lack access control and are registered in the `SmartLoanDiamondBeacon` contract. As a result, any prime account owner can invoke these functions to manipulate the protocol exposure. Specifically, a malicious prime account owner could:

- invoke the `resetPrimeAccountAssetsExposure` function to intentionally lower protocol exposure for certain assets, resulting in inaccurate records.
- repeatedly invoke the `setPrimeAccountAssetsExposure` function to increase the exposure of certain assets (i.e., `exposure.current`) to the `exposure.max` value, which could lead to a DoS issue for some functionality such as `fund` and `borrow`.

```
13  function resetPrimeAccountAssetsExposure() external {
14      bytes32[] memory ownedAssets = DeploymentConstants.getAllOwnedAssets();
15      IStakingPositions.StakedPosition[] storage positions = DiamondStorageLib.stakedPositions();
16      ITokenManager tokenManager = DeploymentConstants.getTokenManager();
17
18      for(uint i=0; i<ownedAssets.length; i++){
19          IERC20Metadata token = IERC20Metadata(tokenManager.getAssetAddress(ownedAssets[i], true)
                );
20          tokenManager.decreaseProtocolExposure(ownedAssets[i], token.balanceOf(address(this)) * 1
                e18 / 10**token.decimals());
21      }
22      for(uint i=0; i<positions.length; i++){
23          (bool success, bytes memory result) = address(this).staticcall(abi.encodeWithSelector(
                positions[i].balanceSelector));
24          if (success) {
25              uint256 balance = abi.decode(result, (uint256));
26              uint256 decimals = IERC20Metadata(tokenManager.getAssetAddress(positions[i].symbol,
                    true)).decimals();
27              tokenManager.decreaseProtocolExposure(positions[i].identifier, balance * 1e18 / 10**
                    decimals);
28          }
```

```
29      }
30  }
31
32  function setPrimeAccountAssetsExposure() external {
33      bytes32[] memory ownedAssets = DeploymentConstants.getAllOwnedAssets();
34      IStakingPositions.StakedPosition[] storage positions = DiamondStorageLib.stakedPositions();
35      ITokenManager tokenManager = DeploymentConstants.getTokenManager();
36
37      for(uint i=0; i<ownedAssets.length; i++){
38          IERC20Metadata token = IERC20Metadata(tokenManager.getAssetAddress(ownedAssets[i], true)
                );
39          tokenManager.increaseProtocolExposure(ownedAssets[i], token.balanceOf(address(this)) * 1
                e18 / 10**token.decimals());
40      }
41      for(uint i=0; i<positions.length; i++){
42          (bool success, bytes memory result) = address(this).staticcall(abi.encodeWithSelector(
                positions[i].balanceSelector));
43          if (success) {
44              uint256 balance = abi.decode(result, (uint256));
45              uint256 decimals = IERC20Metadata(tokenManager.getAssetAddress(positions[i].symbol,
                    true)).decimals();
46              tokenManager.increaseProtocolExposure(positions[i].identifier, balance * 1e18 / 10**
                    decimals);
47          }
48      }
49  }
```

Listing 2.20: contracts/facets/AssetExposureController.sol

Additionally, the above vulnerability introduces an attack vector while interacting with the GmxV2 protocol. Specifically, when GMX keepers finalize a deposit or withdrawal, they invoke the prime account's callback logic (in the `GmxV2CallbacksFacet` contract). The core logic of the callback involves adjusting the asset exposure and unfreezing the prime account. If the callback execution reverts, the revert is caught in a try-catch block, allowing the keeper to proceed with the remaining tasks.

However, a malicious actor can inflate the protocol exposure (via the aforementioned vulnerability) before the callback execution, causing the protocol exposure to reach its maximum limit. This results in the callback reverting when it attempts to adjust the protocol exposure based on the amount of received tokens. Consequently, the failed callback prevents the prime account from being unfrozen, causing a temporary DoS for the affected user. The user would then need to wait for whitelist liquidators to manually unfreeze their accounts.

```
91  function afterDepositCancellation(bytes32 key, Deposit.Props memory deposit, EventUtils.
        EventLogData memory eventData) external onlyGmxV2Keeper nonReentrant override {
92  // ...
93
94  if(deposit.numbers.initialLongTokenAmount > 0) {
95      tokenManager.increaseProtocolExposure(
96          tokenManager.tokenAddressToSymbol(longToken),
97          deposit.numbers.initialLongTokenAmount * 1e18 / 10**IERC20Metadata(longToken).decimals()
98      );
```

```
 99    }
100    if(deposit.numbers.initialShortTokenAmount > 0) {
101        tokenManager.increaseProtocolExposure(
102            tokenManager.tokenAddressToSymbol(shortToken),
103            deposit.numbers.initialShortTokenAmount * 1e18 / 10**IERC20Metadata(shortToken).decimals
                   ()
104        );
105    }
```

Listing 2.21: contracts/facets/GmxV2CallbacksFacet.sol

Impact Inaccurate protocol exposure and potential DoS issues for certain functionalities.

Suggestion Revise the code logic accordingly.

2.2.9 Potential precision loss in the `getPoolPrice` function

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `sPrime`, the function `getPoolPrice` converts the active ID to a price with 18 decimals and then adjusts it to 8 decimals based on the tokens' decimal places. If `tokenXDecimals < 10 + tokenYDecimals`, the function enters the *else* branch and divides the price by 10 raised to the power of `(10 + tokenYDecimals - tokenXDecimals)`.

However, this may cause precision loss when the difference between `tokenYDecimals` and `tokenXDecimals` is substantial. For instance, when `tokenYDecimals` is 18 and `tokenXDecimals` is 8, `(10 + tokenYDecimals - tokenXDecimals)` is equal to 20, which is likely to reduce the `price` to zero.

```
248    function getPoolPrice() public view returns(uint256) {
249        uint256 price = PriceHelper.convert128x128PriceToDecimal(lbPair.getPriceFromId(lbPair.
               getActiveId()));
250        // price * 1e8 * 1edx / 1edy / 1e18
251        if (tokenXDecimals >= 10 + tokenYDecimals) {
252            price = price * 10 ** (tokenXDecimals - 10 - tokenYDecimals);
253        } else {
254            price = price / 10 ** (10 + tokenYDecimals - tokenXDecimals);
255        }
256        return price;
257    }
```

Listing 2.22: contracts/token/sPrime.sol

Impact The calculated price may not meet expectations.

Suggestion Revise the code logic accordingly.

2.2.10 Incorrect decimal handling in the `getUserValueInTokenY` function

Severity Medium

Status Fixed in `Version 2`

Introduced by Version 1-1

Description In the sPrime contract, the getUserValueInTokenY function calculates the estimated USD value of a user's position by retrieving the pool price through the getPoolPrice function, and then invoking the getUserValueInTokenY function with this price. This second function calls the getLiquidityTokenAmounts function in the sPrimeImpl contract to compute the result. However, this leads to improper calculations due to inconsistent logic in handling token decimals.

Specifically, the price is retrieved from lbPair.getPriceFromId(lbPair.getActiveId()), divided by tokenYDecimals, and then multiplied by tokenXDecimals in the getPoolPrice function. Meanwhile, the getLiquidityTokenAmounts function simply takes the price returned by getPoolPrice and multiplies it by tokenYDecimals. This causes the missing adjustment for tokenXDecimals, resulting in an incorrect binId conversion and an incorrect final result.

```
185   function getUserValueInTokenY(address user) external view returns (uint256) {
186       uint256 poolPrice = getPoolPrice();
187       return getUserValueInTokenY(user, poolPrice);
188   }
```

Listing 2.23: contracts/token/sPrime.sol

```
157   function getUserValueInTokenY(address user, uint256 poolPrice) public view returns (uint256) {
158       (,,,,uint256 centerId, uint256[] memory liquidityMinted) = positionManager.positions(
              getUserTokenId(user));
159       IPositionManager.DepositConfig memory depositConfig = positionManager.getDepositConfig(
              centerId);
160
161       if (depositConfig.depositIds.length != liquidityMinted.length) {
162           revert LengthMismatch();
163       }
164
165       uint256 amountX = 0;
166       uint256 amountY = 0;
167       (bool success, bytes memory result) = implementation.staticcall(abi.encodeWithSignature("
              getLiquidityTokenAmounts(uint256[],uint256[],uint256)", depositConfig.depositIds,
              liquidityMinted, poolPrice));
168
169       if (!success) {
170           revert ProxyCallFailed();
171       }
172
173       (amountX, amountY) = abi.decode(result, (uint256, uint256));
174
175       amountY = amountY + FullMath.mulDiv(amountX, poolPrice * 10 ** tokenYDecimals, 10 ** (8 +
              tokenXDecimals));
176
177       return amountY;
178   }
```

Listing 2.24: contracts/token/sPrime.sol

```
248   function getPoolPrice() public view returns(uint256) {
```

```
249       uint256 price = PriceHelper.convert128x128PriceToDecimal(lbPair.getPriceFromId(lbPair.
              getActiveId()));
250       // price * 1e8 * 1edx / 1edy / 1e18
251       if (tokenXDecimals >= 10 + tokenYDecimals) {
252           price = price * 10 ** (tokenXDecimals - 10 - tokenYDecimals);
253       } else {
254           price = price / 10 ** (10 + tokenYDecimals - tokenXDecimals);
255       }
256       return price;
257   }
```

Listing 2.25: contracts/token/sPrime.sol

```
51   function getLiquidityTokenAmounts(uint256[] memory depositIds, uint256[] memory liquidityMinted
         , uint256 poolPrice) public view returns(uint256 amountX, uint256 amountY) {
52
53       ILBPair lbPair = ISPrimeTraderJoe(sPrime).getLBPair();
54       IERC20Metadata tokenY = IERC20Metadata(address(ISPrimeTraderJoe(sPrime).getTokenY()));
55       poolPrice = FullMath.mulDiv(poolPrice, 10 ** tokenY.decimals(), 1e8);
56
57       uint24 binId = lbPair.getIdFromPrice(PriceHelper.convertDecimalPriceTo128x128(poolPrice));
```

Listing 2.26: contracts/token/sPrimeImpl.sol

Impact The result returned by the `getUserValueInTokenY` function may be incorrect.

Suggestion Revise the code logic accordingly.

2.2.11 Unexpected adjustment to the received `sPrime` amount due to spot price dependency

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `sPrimeUniswap`, the hook function `_afterTokenTransfer` is invoked during `sPrime` transfers to move the corresponding liquidity from the `sPrime` sender (i.e., the input `from`) to the `sPrime` receiver (i.e., the input `to`). Specifically, this function mints a new position based on the `tickLower` and `tickUpper` values determined from the sender's existing position. It then adjusts the received `sPrime` amount (i.e., the variable `total`) based on the liquidity added to the new position. Both minting the new position and adjusting the received `sPrime` amount depend on the pool's spot price.

In this case, a malicious user could manipulate the pool price before the `sPrime` transfers, leading to an unexpected adjustment in the received `sPrime` amount. Although the total received asset value would remain unaffected by the manipulated price, the decrease in the received `sPrime` amount could result in a reduction of voting power. The `sPrime` receiver would then be required to perform an additional deposit transaction to recover their voting power, as shown in the example below:

1. The malicious user manipulates the Prime-WETH pool to decrease the price of the Prime token.

2. The malicious user then transfers `sPrime` to a legitimate user or invokes the `transferFrom` function.
   (a). Due to the manipulated price of the Prime token, only the Prime token is added to the new position, which is determined from the sender's position. As a result, `amountXAdded` of WETH becomes 0 and `amountYAdded` of Prime becomes `amountY`.
   (b). During the `sPrime` amount adjustment, since `amountXAdded` is 0, all asset X (i.e., WETH) is transferred to the receiver and excluded from the calculation of the received `sPrime` amount.

```
785   function _afterTokenTransfer(
786       address from,
787       address to,
788       uint256 amount
789   ) internal virtual override {
790       // ...
791               (
792                   uint256 newTokenId,
793                   ,
794                   uint256 amountXAdded,
795                   uint256 amountYAdded
796               ) = positionManager.mint(
797                       INonfungiblePositionManager.MintParams({
798                           token0: address(getToken0()),
799                           token1: address(getToken1()),
800                           fee: feeTier,
801                           tickLower: tickLower,
802                           tickUpper: tickUpper,
803                           amount0Desired: amountX,
804                           amount1Desired: amountY,
805                           amount0Min: 0,
806                           amount1Min: 0,
807                           recipient: address(this),
808                           deadline: block.timestamp
809                       })
810                   );
811               amountX -= amountXAdded;
812               amountY -= amountYAdded;
813
814               if(getToken0() != tokenX) {
815                   (amountXAdded, amountYAdded) = (amountYAdded, amountXAdded);
816                   (amountX, amountY) = (amountY, amountX);
817               }
818
819               uint256 total = _getTotalInTokenY(amountXAdded, amountYAdded);
820
821               _transferTokens(
822                   address(this),
823                   to,
824                   amountX - amountXAdded,
825                   amountY - amountYAdded
826               );
```

Listing 2.27: contracts/token/sPrimeUniswap.sol

```
244   function _getTotalInTokenY(
245       uint256 amountX,
246       uint256 amountY
247   ) internal view returns (uint256 weight) {
248       uint256 amountXToY = _getTokenYFromTokenX(amountX);
249       weight = amountY + amountXToY;
250   }
```

Listing 2.28: contracts/token/sPrimeUniswap.sol

```
257   function _getTokenYFromTokenX(
258       uint256 amountX
259   ) internal view returns (uint256 amountY) {
260       (, int24 tick, , , , , ) = pool.slot0();
261       amountY = OracleLibrary.getQuoteAtTick(
262           tick,
263           uint128(amountX),
264           address(tokenX),
265           address(tokenY)
266       );
267   }
```

Listing 2.29: contracts/token/sPrimeUniswap.sol

Impact Malicious users can reduce the received `sPrime` amount during transfers, diminishing the receiver's voting power.

Suggestion Implement a slippage check in the `_afterTokenTransfer` function.

2.2.12 Incorrect return value in the `getMaximumTokensReceived` function

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description The function `getMaximumTokensReceived` may return incorrect results when the path length exceeds 2. Specifically, it retrieves `amounts[1]` from the `getAmountsOut` function, assuming it represents the final token amount. However, in `getAmountsOut`, the length of the `amounts` array corresponds to the swap path. For paths involving more than 2 tokens, `amounts[1]` refers to an intermediary token amount rather than the intended output. Notably, while this contract is outside the audit scope, multiple audited contracts inherit from it.

```
158   function getMaximumTokensReceived(uint256 _amountIn, address _soldToken, address _boughtToken)
          public view override returns (uint256) {
159       address[] memory path = getPath(_soldToken, _boughtToken);
160
161       return router.getAmountsOut(_amountIn, path)[1];
162   }
```

Listing 2.30: contracts/integrations/UniswapV2Intermediary.sol

```
169   function getPath(address _token1, address _token2) internal virtual view returns (address[]
          memory) {
170       address[] memory path;
171
172       if (_token1 != getNativeTokenAddress() && _token2 != getNativeTokenAddress()) {
173           path = new address[](3);
174           path[0] = _token1;
175           path[1] = getNativeTokenAddress();
176           path[2] = _token2;
177       } else {
178           path = new address[](2);
179           path[0] = _token1;
180           path[1] = _token2;
181       }
182
183       return path;
184   }
```

Listing 2.31: contracts/integrations/UniswapV2Intermediary.sol

```
62   function getAmountsOut(address factory, uint amountIn, address[] memory path) internal view
         returns (uint[] memory amounts) {
63       require(path.length >= 2, 'UniswapV2Library: INVALID_PATH');
64       amounts = new uint[](path.length);
65       amounts[0] = amountIn;
66       for (uint i; i < path.length - 1; i++) {
67           (uint reserveIn, uint reserveOut) = getReserves(factory, path[i], path[i + 1]);
68           amounts[i + 1] = getAmountOut(amounts[i], reserveIn, reserveOut);
69       }
70   }
```

Listing 2.32: UniswapV2Library.sol

Impact Users invoking this function may receive incorrect results, potentially leading to unexpected outcomes or risks.

Suggestion Fetch the last element in the `amounts` array dynamically based on its length.

2.2.13 Improper implementations violating the withdrawal guard design

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description According to the contract `IndexRouter`, users can specify the recipient of output tokens when minting or burning `CAI` tokens. However, both `mintCai` and `burnCai` functions lack validation on the parameters. Exploiting this, users could specify output token recipients different from the prime account. Although the modifier `remainsSolvent` exists here to prevent

insolvency conditions, this breaks DeltaPrime's design principle of withdrawal guard. Similarly, several functions in the current implementation could be exploited to withdraw assets, violating this design.

```
29  function mintCai(
30      bytes4 selector,
31      bytes memory data,
32      address fromToken,
33      uint256 fromAmount,
34      uint256 maxSlippage
35  ) external nonReentrant onlyOwner remainsSolvent {
36      // ...
37
38      (bool success, ) = INDEX_ROUTER.call((abi.encodePacked(selector, data)));
39      require(success, "Mint failed");
```

Listing 2.33: contracts/facets/avalanche/CaiFacet.sol

```
 7  interface IIndexRouter {
 8      struct MintParams {
 9          address index;
10          uint amountInBase;
11          address recipient;
12      }
```

Listing 2.34: IIndexRouter.sol on Avalanche chain

Additional Case 1: In the function `unstakeFromPenpieAndWithdrawFromPendle` of the contract `PenpieFacet`, the `output` parameter is used in the withdrawal process to perform swaps and thus can be controlled by users to specify recipients during the swap.

```
142  (netTokenOut, , ) = IPendleRouter(PENDLE_ROUTER)
143  .removeLiquiditySingleToken(
144      address(this),
145      market,
146      amount,
147      output,
148      limit
149  );
150  require(netTokenOut >= minOut, "Too little received");
```

Listing 2.35: contracts/facets/arbitrum/PenpieFacet.sol

Additional Case 2: In the function `paraSwapV2` of the contract `ParaSwapFacet` and the function `swapDebtParaSwap` of the contract `AssetsOperationsFacet`, users could specify arbitrary recipients in the `data` parameter.

```
136  function paraSwapV2(
137      bytes4 selector,
138      bytes memory data,
139      address fromToken,
140      uint256 fromAmount,
141      address toToken,
142      uint256 minOut
```

```
143   )
144       external
145       nonReentrant
146       onlyOwner
147       noBorrowInTheSameBlock
148       remainsSolvent
149   {
150       // ...
151
152       (bool success, ) = PARA_ROUTER.call((abi.encodePacked(selector, data)));
153       require(success, "Swap failed");
```

Listing 2.36: contracts/facets/ParaSwapFacet.sol

```
271   function swapDebtParaSwap(bytes32 _fromAsset, bytes32 _toAsset, uint256 _repayAmount, uint256
          _borrowAmount, bytes4 selector, bytes memory data) external onlyOwnerOrInsolvent
          remainsSolvent nonReentrant {
272       // ...
273
274           (bool success, ) = PARA_ROUTER.call((abi.encodePacked(selector, data)));
275           require(success, "Swap failed");
```

Listing 2.37: contracts/facets/AssetsOperationsFacet.sol

Impact Users could withdraw assets without verifying whether they have sufficient assets to repay.

Suggestion Revise the code logic accordingly.

Clarification from BlockSec The project resolved the issue related to the `PenpieFacet` contract by removing its integration with Pendle.

2.2.14 Unclaimable fee in the `UniswapV3Facet` contract

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description The contract `UniswapV3Facet` allows prime accounts to manage liquidity by interacting with the UniswapV3's contract `NonfungiblePositionManager`. Prime accounts can invoke the function `decreaseLiquidityUniswapV3` to remove liquidity. However, the function does not handle fee collection, and the contract does not implement any functions to claim the fee.

Specifically, the function interacts with the contract `NonfungiblePositionManager`, calling the function `decreaseLiquidity` to retrieve the token amounts (i.e., `amount0` and `amount1`), and then passes them as parameters to invoke the function `collect`. Notably, according to the design of UniswapV3, the liquidity fees are just recorded in position, and the values returned by the function `decreaseLiquidity`, `amount0` and `amount1` represent decreased liquidity without including the fees. Additionally, the contract does not include other functions that can invoke `NonfungiblePositionManager`'s function `collect`. This causes prime accounts to fail to claim their liquidity fees.

```
165   function decreaseLiquidityUniswapV3(INonfungiblePositionManager.DecreaseLiquidityParams memory
          params) external nonReentrant noBorrowInTheSameBlock onlyOwnerOrInsolvent {
166       // ...
167
168       (
169           uint256 amount0,
170           uint256 amount1
171       ) = INonfungiblePositionManager(NONFUNGIBLE_POSITION_MANAGER_ADDRESS).decreaseLiquidity(
              params);
172
173       //TODO: check risks of uint256 to uint128 conversion
174       INonfungiblePositionManager.CollectParams memory collectParams =
              INonfungiblePositionManager.CollectParams(params.tokenId, address(this), uint128(
              amount0), uint128(amount1));
175
176       INonfungiblePositionManager(NONFUNGIBLE_POSITION_MANAGER_ADDRESS).collect(collectParams);
```

Listing 2.38: contracts/facets/avalanche/UniswapV3Facet.sol

Impact Prime accounts cannot claim their liquidity fees.

Suggestion Add a function that allows users to collect or withdraw accumulated fees.

2.2.15 Unenforced modifier `noBorrowInTheSameBlock` in specific functions

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `SolvencyMethods`, the modifier `noBorrowInTheSameBlock` checks that `ds._lastBorrowTimestamp` is not equal to `block.timestamp`, ensuring borrowing occurs in a standalone transaction. However, users can bypass this modifier check by invoking the function `swapDebtParaSwap` of the contract `AssetsOperationsFacet` to conduct borrows. Specifically, with customized inputs (i.e., setting `_repayAmount` to zero), the function `swapDebtParaSwap` behaves identically to the function `borrow` but without updating `ds._lastBorrowTimestamp`. A similar issue occurs in the function `swapDebt` as well.

```
301   modifier noBorrowInTheSameBlock() {
302       DiamondStorageLib.DiamondStorage storage ds = DiamondStorageLib.diamondStorage();
303       require(ds._lastBorrowTimestamp != block.timestamp, "Borrowing must happen in a standalone
              transaction");
304       _;
305   }
```

Listing 2.39: contracts/lib/SolvencyMethods.sol

```
271   function swapDebtParaSwap(bytes32 _fromAsset, bytes32 _toAsset, uint256 _repayAmount, uint256
          _borrowAmount, bytes4 selector, bytes memory data) external onlyOwnerOrInsolvent
          remainsSolvent nonReentrant {
272       ITokenManager tokenManager = DeploymentConstants.getTokenManager();
273       Pool fromAssetPool = Pool(tokenManager.getPoolAddress(_fromAsset));
```

```
274         _repayAmount = Math.min(_repayAmount, fromAssetPool.getBorrowed(address(this)));
275
276         IERC20Metadata toToken = getERC20TokenInstance(_toAsset, false);
277         IERC20Metadata fromToken = getERC20TokenInstance(_fromAsset, false);
278
279         Pool toAssetPool = Pool(tokenManager.getPoolAddress(_toAsset));
280         toAssetPool.borrow(_borrowAmount);
281
282         uint256 initialRepayTokenAmount = fromToken.balanceOf(address(this));
283
284         {
285
286             // swap toAsset to fromAsset
287             address(toToken).safeApprove(PARA_TRANSFER_PROXY, 0);
288             address(toToken).safeApprove(PARA_TRANSFER_PROXY, _borrowAmount);
289
290             (bool success, ) = PARA_ROUTER.call((abi.encodePacked(selector, data)));
291             require(success, "Swap failed");
292
293         }
294         _repayAmount = Math.min(fromToken.balanceOf(address(this)), _repayAmount);
295
296         _processRepay(tokenManager, fromAssetPool, address(fromToken), _repayAmount, fromToken.
                balanceOf(address(this)) - initialRepayTokenAmount);
297
298         emit DebtSwap(msg.sender, address(fromToken), address(toToken), _repayAmount, _borrowAmount
                , block.timestamp);
299     }
```

Listing 2.40: contracts/facets/AssetsOperationsFacet.sol

```
80  function _processRepay(ITokenManager tokenManager, Pool fromAssetPool, address fromToken,
        uint256 repayAmount, uint256 receivedRepayTokenAmount) internal {
81      fromToken.safeApprove(address(fromAssetPool), 0);
82      fromToken.safeApprove(address(fromAssetPool), repayAmount);
83      fromAssetPool.repay(repayAmount);
```

Listing 2.41: contracts/facets/AssetsOperationsFacet.sol

Impact The modifier `noBorrowInTheSameBlock` is not enforced in some functions, allowing the check to be bypassed.

Suggestion Revise the code logic accordingly.

2.2.16 Flawed design allowing stealing of assets from insolvent prime accounts

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description This protocol allows liquidators (via the modifiers `onlyWhitelistedLiquidators` or `onlyOwnerOrInsolvent`) to perform asset conversions for insolvent prime accounts without

a solvency check. However, this design enables malicious liquidators to siphon assets from insolvent prime accounts in the following two cases:

1. **Case 1 (the `ParaSwapFacet` contract):** The function `paraSwapBeforeLiquidation` is used for asset conversion before liquidations and is only available to liquidators. Additionally, the function includes a 2% slippage check to prevent unreasonable swaps.
   However, this design allows malicious liquidators to exploit the 2% slippage tolerance for profit. Specifically, with the customized input `data`, a malicious liquidator can perform an external call to their customized contract (i.e., `callees`) via the contract `PARA_ROUTER`. In the external call, the liquidator could conduct a swap and transfer a portion of the received tokens to the prime account to bypass the 2% slippage check, thereby making a profit. Finally, the liquidator could repeatedly leverage this method to drain all funds from insolvent prime accounts.

```
74  function paraSwapBeforeLiquidation(
75      bytes4 selector,
76      bytes memory data,
77      address fromToken,
78      uint256 fromAmount,
79      address toToken,
80      uint256 minOut
81  )
82  external
83  nonReentrant
84  onlyWhitelistedLiquidators
85  noBorrowInTheSameBlock
86  {
87      require(!_isSolvent(), "Cannot perform on a solvent account");
88
89      // ...
90
91      (bool success, ) = PARA_ROUTER.call((abi.encodePacked(selector, data)));
92      require(success, "Swap failed");
93
94      uint256 boughtTokenFinalAmount = swapTokensDetails.boughtToken.balanceOf(
95          address(this)
96      ) - swapTokensDetails.initialBoughtTokenBalance;
97      require(boughtTokenFinalAmount >= minOut, "Too little received");
98
99      uint256 soldTokenFinalAmount = swapTokensDetails.initialSoldTokenBalance -
            swapTokensDetails.soldToken.balanceOf(
100         address(this)
101     );
102     require(soldTokenFinalAmount == fromAmount, "Too much sold");
103
104     // ...
105
106     uint256 soldTokenDollarValue = prices[0] * soldTokenFinalAmount * 10**10 / 10 **
            swapTokensDetails.soldToken.decimals();
107     uint256 boughtTokenDollarValue = prices[1] * boughtTokenFinalAmount * 10**10 / 10 **
            swapTokensDetails.boughtToken.decimals();
108     if(soldTokenDollarValue > boughtTokenDollarValue) {
```

```
109         // If the sold token is more valuable than the bought token, we need to check the
                slippage
110         // If the slippage is too high, we revert the transaction
111         // Slippage = (soldTokenDollarValue - boughtTokenDollarValue) * 100 /
                soldTokenDollarValue
112         uint256 slippage = (soldTokenDollarValue - boughtTokenDollarValue) * 100 /
                soldTokenDollarValue;
113         require(slippage < 2, "Slippage too high"); // MAX 2% slippage
114     }
115 }
```

Listing 2.42: contracts/facets/ParaSwapFacet.sol

2. **Case 2 (the `PenpieFacet` contract):** The `unstakeFromPenpieAndWithdrawFromPendle` function allows liquidators to remove liquidity from the Pendle protocol for insolvent prime accounts. However, by providing customized input (i.e., `output`), a malicious liquidator could specify the receiver of the removed liquidity. As a result, the malicious liquidator can siphon all staked liquidity from insolvent prime accounts to their own EOAs.

```
108 function unstakeFromPenpieAndWithdrawFromPendle(
109     bytes32 asset,
110     uint256 amount,
111     address market,
112     uint256 minOut,
113     IPendleRouter.TokenOutput memory output,
114     IPendleRouter.LimitOrderData memory limit
115 ) external onlyOwnerOrInsolvent nonReentrant returns (uint256) {
116     // ...
117
118         (netTokenOut, , ) = IPendleRouter(PENDLE_ROUTER)
119             .removeLiquiditySingleToken(
120                 address(this),
121                 market,
122                 amount,
123                 output,
124                 limit
125             );
```

Listing 2.43: contracts/facets/arbitrum/PenpieFacet.sol

Impact Malicious liquidators can drain assets from insolvent prime accounts to their EOAs.

Suggestion Revise the code logic accordingly.

Clarification from BlockSec The project resolved the issue related to the `PenpieFacet` contract by removing its integration with Pendle.

2.2.17 Potential bypass of whitelisted borrower limitation

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `SmartLoansFactoryRestrictedAccess`, borrowers who are not on the whitelist are prohibited from creating a prime account. However, whitelisted borrowers can create a prime account and transfer it to other users, who are not on the whitelist or have been delisted, via the function `changeOwnership`.

```
36  function createLoan() public virtual override hasNoLoan canCreatePrimeAccount(msg.sender)
        returns (SmartLoanDiamondBeacon) {
37      return super.createLoan();
38  }
39
40  function createAndFundLoan(bytes32 _fundedAsset, uint256 _amount) public virtual override
        hasNoLoan canCreatePrimeAccount(msg.sender) returns (SmartLoanDiamondBeacon) {
41      return super.createAndFundLoan(_fundedAsset, _amount);
42  }
```

Listing 2.44: contracts/SmartLoansFactoryRestrictedAccess.sol

```
60  modifier canCreatePrimeAccount(address _borrower) {
61      require(isBorrowerWhitelisted(_borrower), "Only whitelisted borrowers can create a Prime
            Account.");
62      _;
63  }
```

Listing 2.45: contracts/SmartLoansFactoryRestrictedAccess.sol

```
48  function changeOwnership(address _newOwner) public {
49      address loan = msg.sender;
50      address oldOwner = loansToOwners[loan];
51
52      require(oldOwner != address(0), "Only a SmartLoan can change it's owner");
53      require(!_hasLoan(_newOwner), "New owner already has a loan");
54
55      ownersToLoans[oldOwner] = address(0);
56      ownersToLoans[_newOwner] = loan;
57      loansToOwners[loan] = _newOwner;
58  }
```

Listing 2.46: contracts/SmartLoansFactory.sol

Impact Borrowers who are not on the whitelist can obtain a prime account.

Suggestion Revise the code logic accordingly.

Feedback from the project The project indicated that the `SmartLoansFactoryRestrictedAccess` contract was only used during the first alpha deployment of the protocol, and they have since removed this contract.

2.2.18 Potential DoS due to the lack of pop operation for `locks`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `Pool`, the variable `locks` records the locking details for accounts. However, there is only a push operation for the array `locks[account]`, which continuously increases its size. This design could lead to potential failures during iteration due to gas exhaustion. Consequently, the functions `lockDeposit`, `withdraw`, `transfer`, and `transferFrom` might face potential DoS issues.

```
95   function lockDeposit(uint256 amount, uint256 lockTime) external gated {
96       if (getNotLockedBalance(msg.sender) < amount) {
97           revert InsufficientBalanceToLock();
98       }
99       if (lockTime > MAX_LOCK_TIME) {
100          revert LockTimeExceedsMax();
101      }
102      locks[msg.sender].push(LockDetails(lockTime, amount, block.timestamp + lockTime));
```

Listing 2.47: contracts/Pool.sol

```
74   function getLockedBalance(address account) public view returns (uint256) {
75       uint256 lockedBalance = 0;
76       for (uint i = 0; i < locks[account].length; i++) {
77           if (locks[account][i].unlockTime > block.timestamp) {
78               lockedBalance += locks[account][i].amount;
79           }
80       }
81       return lockedBalance;
82   }
83
84   function getNotLockedBalance(address account) public view returns (uint256 notLockedBalance) {
85       uint256 lockedBalance = getLockedBalance(account);
86       uint256 balance = balanceOf(account);
87       if(balance < lockedBalance) {
88           notLockedBalance = 0;
89       } else {
90           return balance - lockedBalance;
91       }
92   }
```

Listing 2.48: contracts/Pool.sol

```
443  function isWithdrawalAmountAvailable(address account, uint256 amount) public view returns (bool
         ) {
444      return amount <= getNotLockedBalance(account);
445  }
```

Listing 2.49: contracts/Pool.sol

```
451  function withdraw(uint256 _amount) external nonReentrant gated {
452      if (!isWithdrawalAmountAvailable(msg.sender, _amount)) {
453          revert BalanceLocked();
454      }
```

Listing 2.50: contracts/Pool.sol

Impact Users are unable to transfer and withdraw their funds.

Suggestion Remove the outdated elements from the array `locks[account]`.

2.2.19 Potential DoS due to incomplete removal of unsupported assets

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `AssetsOperationsFacet`, the `removeUnsupportedOwnedAsset` function is used to remove unsupported assets, whose corresponding records have been cleaned in the contract `TokenManager`. Specifically, in the `removeUnsupportedOwnedAsset` function, the states (`tokenToStatus`, `tokenAddressToSymbol`, `debtCoverage`, and `identifierToExposureGroup`) of the unsupported assets in the contract `TokenManager` are required to be set as zero before the removal. However, cleaning the corresponding states requires invoking both functions `removeTokenAsset` and `setIdentifiersToExposureGroups` in the contract `TokenManager`.

This design might lead to a potential DoS issue due to an incomplete cleaning of corresponding states in the contract `TokenManager`. Specifically, based on the protocol design, most operations from the prime account require a solvency check. If the owner only invokes `removeTokenAsset`, then all invocations of `removeUnsupportedOwnedAsset` would fail, and all operations requiring a solvency check might revert (due to a non-existent asset in the contract `TokenManager`) before invoking `setIdentifiersToExposureGroups`.

```
35  function removeUnsupportedOwnedAsset(bytes32 _asset, address _address) external
        onlyWhitelistedLiquidators nonReentrant {
36      ITokenManager tokenManager = DeploymentConstants.getTokenManager();
37
38      // Check if the asset exists in the TokenManager
39      require(tokenManager.tokenToStatus(_address) == 0, "Asset is still supported");
40      require(tokenManager.tokenAddressToSymbol(_address) == bytes32(0), "Asset address to symbol
            not empty");
41      require(tokenManager.debtCoverage(_address) == 0, "Asset still has debt coverage");
42      require(tokenManager.identifierToExposureGroup(_asset) == bytes32(0), "Asset still has
            exposure group");
43
44      bytes32[] memory allAssets = tokenManager.getAllTokenAssets();
45      // Loop through all assets and check if the asset exists
46      for (uint i = 0; i < allAssets.length; i++) {
47          require(allAssets[i] != _asset, "Asset exists in TokenManager");
48      }
```

Listing 2.51: contracts/facets/AssetsOperationsFacet.sol

```
251 function removeTokenAssets(bytes32[] memory _tokenAssets) public onlyOwner {
252     for (uint256 i = 0; i < _tokenAssets.length; i++) {
253         _removeTokenAsset(_tokenAssets[i]);
254     }
255 }
256
```

```
257   function _removeTokenAsset(bytes32 _tokenAsset) internal {
258       address tokenAddress = getAssetAddress(_tokenAsset, true);
259       EnumerableMap.remove(assetToTokenAddress, _tokenAsset);
260       tokenAddressToSymbol[tokenAddress] = 0;
261       tokenToStatus[tokenAddress] = _NOT_SUPPORTED;
262       debtCoverage[tokenAddress] = 0;
263       _removeTokenFromList(tokenAddress);
264       emit TokenAssetRemoved(msg.sender, _tokenAsset, block.timestamp);
265   }
```

Listing 2.52: contracts/TokenManager.sol

```
288   function _getTWVOwnedAssets(AssetPrice[] memory ownedAssetsPrices) internal virtual view
          returns (uint256) {
289       bytes32 nativeTokenSymbol = DeploymentConstants.getNativeTokenSymbol();
290       ITokenManager tokenManager = DeploymentConstants.getTokenManager();
291
292       uint256 weightedValueOfTokens = ownedAssetsPrices[0].price * address(this).balance *
              tokenManager.debtCoverage(tokenManager.getAssetAddress(nativeTokenSymbol, true)) / (10
              ** 26);
293
294       if (ownedAssetsPrices.length > 0) {
295
296           for (uint256 i = 0; i < ownedAssetsPrices.length; i++) {
297               IERC20Metadata token = IERC20Metadata(tokenManager.getAssetAddress(ownedAssetsPrices
                      [i].asset, true));
298               weightedValueOfTokens = weightedValueOfTokens + (ownedAssetsPrices[i].price * token.
                      balanceOf(address(this)) * tokenManager.debtCoverage(address(token)) / (10 **
                      token.decimals() * 1e8));
299           }
300       }
301       return weightedValueOfTokens;
302   }
```

Listing 2.53: contracts/facets/SolvencyFacetProd.sol

```
 98   function getAssetAddress(bytes32 _asset, bool allowInactive) public view returns (address) {
 99       (, address assetAddress) = assetToTokenAddress.tryGet(_asset);
100       require(assetAddress != address(0), "Asset not supported.");
101       if (!allowInactive) {
102           require(tokenToStatus[assetAddress] == _ACTIVE, "Asset inactive");
103       }
104
105       return assetAddress;
106   }
```

Listing 2.54: contracts/TokenManager.sol

Impact Potential DoS for prime account operations due to failure to remove unsupported assets.

Suggestion Revise the code logic accordingly.

2.2.20 Unfair reward distribution due to inconsistent deposit records

Severity Low

Status Confirmed

Introduced by `Version 1-1`

Description The contract `depositRewarder` is designed to distribute rewards based on user deposits. Specifically, both functions `stakeFor` and `addDeposits` are used to update deposit records (i.e., `balanceOf` in the contract `depositRewarder`). However, the `addDeposits` function retrieves the user's deposit amount through `IERC20(pool).balanceOf(account)`, which includes the accrued interest. As a result, users whose deposit amount is recorded via the `addDeposits` function might receive a larger share of the rewards.

```
133   function addDeposits(address[] memory accounts) external onlyOwner {
134       rewardPerTokenStored = rewardPerToken();
135       updatedAt = lastTimeRewardApplicable();
136
137       uint256 length = accounts.length;
138       for (uint256 i; i != length; ++i) {
139           address account = accounts[i];
140           rewards[account] = earned(account);
141           uint256 balance = IERC20(pool).balanceOf(account);
142           uint256 oldBalance = balanceOf[account];
143           balanceOf[account] = balance;
144           totalSupply = totalSupply + balance - oldBalance;
145           userRewardPerTokenPaid[account] = rewardPerTokenStored;
146       }
147
148       emit BatchDeposited(accounts, block.timestamp);
149   }
```

Listing 2.55: contracts/DepositRewarderAbstract.sol

```
151   function stakeFor(
152       uint256 amount,
153       address account
154   ) external nonReentrant onlyPool updateReward(account) {
155       balanceOf[account] += amount;
156       totalSupply += amount;
157
158       emit Deposited(account, amount, block.timestamp);
159   }
```

Listing 2.56: contracts/DepositRewarderAbstract.sol

```
596   function balanceOf(address user) public view override returns (uint256) {
597       return depositIndex.getIndexedValue(_deposited[user], user);
598   }
```

Listing 2.57: contracts/Pool.sol

Impact Accrued interest is recorded in the variable `balanceOf`, resulting in an unfair reward distribution.

Suggestion Revise the code logic accordingly.

Feedback from the project The project stated that the `addDeposits` function is only used during contract initialization, when setting up existing depositors and their balances. From that point forward, all depositor updates are handled exclusively through the `stakeFor` and `withdrawFrom` functions. Additionally, any potential discrepancy arising from the initial addition of deposit interest is minimal.

2.2.21 Lack of check on the asset distinction when swapping debts

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `AssetsOperationsFacet`, both the `swapDebt` and `swapDebtParaSwap` functions allow users to swap debts from one asset (i.e., `_fromAsset`) to another (i.e., `_toAsset`). However, there is no validation to ensure that `_fromAsset` and `_toAsset` are distinct. This lack of validation could lead to an incorrect decrease of the protocol's exposure.

Specifically, if `_fromAsset` and `_toAsset` refer to the same asset, the process involves borrowing `_fromAsset` and then immediately repaying it. In this scenario, there is no change in the balance for `_fromAsset`. However, the repaid amount is treated as a repayment of the previous debt. Consequently, when the function `_processRepay` is called, `repayAmount` may exceed the `receivedRepayTokenAmount`, resulting in an incorrect decrease of the protocol's exposure.

```
233   function swapDebt(bytes32 _fromAsset, bytes32 _toAsset, uint256 _repayAmount, uint256
          _borrowAmount, address[] calldata _path, address[] calldata _adapters) external onlyOwner
          remainsSolvent nonReentrant {
234       ITokenManager tokenManager = DeploymentConstants.getTokenManager();
235       Pool fromAssetPool = Pool(tokenManager.getPoolAddress(_fromAsset));
236       _repayAmount = Math.min(_repayAmount, fromAssetPool.getBorrowed(address(this)));
237
238       IERC20Metadata toToken = getERC20TokenInstance(_toAsset, false);
239       IERC20Metadata fromToken = getERC20TokenInstance(_fromAsset, false);
240
241       require(address(toToken) == _path[0], "Invalid token input");
242       require(address(fromToken) == _path[_path.length - 1], "Invalid token input");
243
244       Pool(tokenManager.getPoolAddress(_toAsset)).borrow(_borrowAmount);
245       uint256 initialRepayTokenAmount = fromToken.balanceOf(address(this));
246
247       {
248           // swap toAsset to fromAsset
249           address(toToken).safeApprove(YY_ROUTER(), 0);
250           address(toToken).safeApprove(YY_ROUTER(), _borrowAmount);
251
252           IYieldYakRouter router = IYieldYakRouter(YY_ROUTER());
253
254           IYieldYakRouter.Trade memory trade = IYieldYakRouter.Trade({
255               amountIn: _borrowAmount,
256               amountOut: _repayAmount,
```

```
257             path: _path,
258             adapters: _adapters
259         });
260
261         router.swapNoSplit(trade, address(this), 0);
262     }
263
264     _repayAmount = Math.min(_repayAmount, fromToken.balanceOf(address(this)));
265
266     _processRepay(tokenManager, fromAssetPool, address(fromToken), _repayAmount, fromToken.
            balanceOf(address(this)) - initialRepayTokenAmount);
267
268     emit DebtSwap(msg.sender, address(fromToken), address(toToken), _repayAmount, _borrowAmount
            , block.timestamp);
269 }
```

Listing 2.58: contracts/facets/AssetsOperationsFacet.sol

```
80  function _processRepay(ITokenManager tokenManager, Pool fromAssetPool, address fromToken,
        uint256 repayAmount, uint256 receivedRepayTokenAmount) internal {
81      fromToken.safeApprove(address(fromAssetPool), 0);
82      fromToken.safeApprove(address(fromAssetPool), repayAmount);
83      fromAssetPool.repay(repayAmount);
84
85      if(receivedRepayTokenAmount > repayAmount) {
86          _increaseExposure(tokenManager, fromToken, receivedRepayTokenAmount - repayAmount);
87      } else {
88          _decreaseExposure(tokenManager, fromToken, repayAmount - receivedRepayTokenAmount);
89      }
90  }
```

Listing 2.59: contracts/facets/AssetsOperationsFacet.sol

Impact Malicious actors could exploit this issue to bypass the maximum protocol exposure restriction.

Suggestion Add a check for the parameters `_fromAsset` and `_toAsset` to ensure they are distinct.

2.2.22 Incorrect return value due to unsorted `depositForm`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `sPrimeImpl`, the function `binInRange` checks whether the active ID is within the user's position range (i.e., `depositIds`), which is assumed to be in ascending order. When a user invokes the `deposit` function in the `sPrime` contract for the first time, the position and corresponding range are initialized in the `PositionManager` contract based on a predefined list, `depositForm`. However, `depositForm` is set based on the input `depositForm_` in the `initialize` function of the `sPrime` contract, and there is no guarantee that `depositForm_`

is sorted. An unsorted `depositForm` could lead to the creation of an unsorted position range, resulting in an incorrect return value from the `binInRange` function in the `sPrimeImpl` contract.

```
31  function binInRange(uint256 tokenId) public view returns(bool) {
32      IPositionManager positionManager = ISPrimeTraderJoe(sPrime).positionManager();
33      ILBPair lbPair = ISPrimeTraderJoe(sPrime).getLBPair();
34
35      IPositionManager.DepositConfig memory depositConfig = positionManager.
            getDepositConfigFromTokenId(tokenId);
36
37      uint256[] memory depositIds = depositConfig.depositIds;
38      uint256 activeId = lbPair.getActiveId();
39      if (depositIds[0] <= activeId && depositIds[depositIds.length - 1] >= activeId) {
40          return true;
41      }
42      return false;
43  }
```

Listing 2.60: contracts/token/sPrimeImpl.sol

```
332 function _encodeDepositConfigs(uint256 centerId) internal view returns (bytes32[] memory
        liquidityConfigs, uint256[] memory depositIds) {
333     uint256 length = depositForm.length;
334     liquidityConfigs = new bytes32[](length);
335     depositIds = new uint256[](length);
336     for (uint256 i = 0; i < length; ++i) {
337         DepositForm memory config = depositForm[i];
338         int256 _id = int256(centerId) + config.deltaId;
339         if (!(_id >= 0 && uint256(_id) <= type(uint24).max)) {
340             revert Overflow();
341         }
342         depositIds[i] = uint256(_id);
343         liquidityConfigs[i] = LiquidityConfigurations.encodeParams(config.distributionX, config.
                distributionY, uint24(uint256(_id)));
344     }
345 }
```

Listing 2.61: contracts/token/sPrime.sol

```
68  function initialize(address tokenX_, address tokenY_, string memory name_, DepositForm[]
        calldata depositForm_, IPositionManager positionManager_, address traderJoeV2Router_)
        external initializer {
69      __PendingOwnable_init();
70      __ReentrancyGuard_init();
71      __ERC20_init(name_, "sPrime");
72
73      traderJoeV2Router = traderJoeV2Router_;
74      ILBFactory lbFactory = ILBRouter(traderJoeV2Router).getFactory();
75      ILBFactory.LBPairInformation memory pairInfo = lbFactory.getLBPairInformation(IERC20(
            tokenX_), IERC20(tokenY_), DEFAULT_BIN_STEP);
76
77      lbPair = pairInfo.LBPair;
78      tokenX = IERC20Metadata(address(lbPair.getTokenX()));
79      tokenY = IERC20Metadata(address(lbPair.getTokenY()));
```

```
80
81      tokenXDecimals = tokenX.decimals();
82      tokenYDecimals = tokenY.decimals();
83
84      for(uint256 i = 0 ; i < depositForm_.length ; i ++) {
85          depositForm.push(depositForm_[i]);
86      }
87
88      positionManager = positionManager_;
89  }
```

Listing 2.62: contracts/token/sPrime.sol

Impact The function `binInRange` may return an incorrect status.

Suggestion Revise the code logic accordingly.

2.2.23 Unhandled native asset in the `claimReward` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `TraderJoeV2Facet`, the native token is not handled in the function `claimReward` on line#71, in contrast to the function `claimReward` on line#91. This inconsistency could lead to potential issues, including a DoS, if the input `merkleEntries` includes a native token.

```
71  function claimReward(IRewarder.MerkleEntry[] calldata merkleEntries) external nonReentrant
        onlyOwner {
72      uint256 length = merkleEntries.length;
73      IERC20[] memory tokens = new IERC20[](length);
74      uint256[] memory beforeBalances = new uint256[](length);
75      for (uint256 i; i != length; ++i) {
76          tokens[i] = merkleEntries[i].token;
77          beforeBalances[i] = tokens[i].balanceOf(address(this));
78      }
79
80      IRewarder rewarder = IRewarder(REWARDER);
81      rewarder.batchClaim(merkleEntries);
82
83      for (uint256 i; i != length; ++i) {
84          uint256 newBalance = tokens[i].balanceOf(address(this));
85          if (newBalance > beforeBalances[i]) {
86              address(tokens[i]).safeTransfer(msg.sender, newBalance - beforeBalances[i]);
87          }
88      }
89  }
```

Listing 2.63: contracts/TraderJoeV2Facet.sol

```
91  function claimReward(ILBPair pair, uint256[] calldata ids) external nonReentrant onlyOwner {
92      ILBHookLens lbHookLens = ILBHookLens(getJoeV2LBHookLens());
```

```
93      ILBHookLens.Parameters memory hookLens = lbHookLens.getHooks(address(pair));
94      address baseRewarder = hookLens.hooks;
95
96
97      if(baseRewarder == address(0)) revert TraderJoeV2NoRewardHook();
98
99
100     address rewardToken = address(ILBHooksBaseRewarder(baseRewarder).getRewardToken());
101     bool isNative = (rewardToken == address(0));
102     uint256 beforeBalance = isNative ? address(this).balance : IERC20(rewardToken).balanceOf(
            address(this));
103     ILBHooksBaseRewarder(baseRewarder).claim(address(this), ids);
104     uint256 reward = isNative ? address(this).balance - beforeBalance : IERC20(rewardToken).
            balanceOf(address(this)) - beforeBalance;
105     if(reward > 0) {
106         if(isNative) {
107             payable(msg.sender).safeTransferETH(reward);
108         } else {
109             rewardToken.safeTransfer(msg.sender, reward);
110         }
111     }
112 }
```

Listing 2.64: contracts/TraderJoeV2Facet.sol

Impact Potential DoS due to missing logic for handling the native token in the `claimReward` function.

Suggestion Add the necessary code logic to ensure proper handling of the native token.

2.2.24 Double-counting issue due to the lack of duplication check for the variable `whitelistedSPrimeContracts`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `vPrimeController`, the variable `whitelistedSPrimeContracts` stores a list of whitelisted `sPrime` contracts for value calculation. In the functions`initialize` and `updateWhitelistedSPrimeContracts` functions, the inputs (i.e., `_whitelistedSPrimeContracts` and `newWhitelistedSPrimeContracts`) are assigned directly to `whitelistedSPrimeContracts` without checking for duplication. This omission could result in incorrect value calculations in the `getUserSPrimeDollarValueVestedAndNonVested` function, where the dollar value of duplicated `sPrime` contracts may be double-counted. Such double-counting could cause an unexpected increase in `vPrimeBalanceLimit` and `vPrimeBalanceDelta` in the `getUserVPrimeRateAndMaxCap` function, leading to incorrect updates by the `updateVPrimeSnapshot` function in the `vPrime` contract for users.

```
48  function initialize(ISPrime[] memory _whitelistedSPrimeContracts, ITokenManager _tokenManager,
        vPrime _vPrime, bool _useOraclePrimeFeed) external initializer {
49      whitelistedSPrimeContracts = _whitelistedSPrimeContracts;
```

```
50      tokenManager = _tokenManager;
51      vPrimeContract = _vPrime;
52      useOraclePrimeFeed = _useOraclePrimeFeed;
53      __PendingOwnable_init();
54  }
```

Listing 2.65: contracts/token/vPrimeController.sol

```
118 function updateWhitelistedSPrimeContracts(ISPrime[] memory newWhitelistedSPrimeContracts)
        external onlyOwner {
119     whitelistedSPrimeContracts = newWhitelistedSPrimeContracts;
120     emit WhitelistedSPrimeContractsUpdated(newWhitelistedSPrimeContracts, msg.sender, block.
            timestamp);
121 }
```

Listing 2.66: contracts/token/vPrimeController.sol

```
60  function updateVPrimeSnapshot(address userAddress) public {
61      (int256 vPrimeRate, uint256 vPrimeBalanceLimit, uint256 alreadyVestedVPrimeBalance) =
            getUserVPrimeRateAndMaxCap(userAddress);
```

Listing 2.67: contracts/token/vPrimeController.sol

Impact May lead to incorrect updates in the `updateVPrimeSnapshot` function.

Suggestion Add a duplication check when assigning or updating the variable.

2.2.25 Temporary DoS in the `fundLiquidityTraderJoeV2` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `TraderJoeV2Facet` contract, the `fundLiquidityTraderJoeV2` function funds LP tokens to the TraderJoeV2 protocol for a prime account. Each prime account has a limited number of bins, determined by the `maxBinsPerPrimeAccount` function. However, the lack of access control in `fundLiquidityTraderJoeV2` allows malicious actors to occupy a prime account's bins by repeatedly adding minimal liquidity with unused `ids`. This design flaw could result in a temporary DoS, as legitimate users must manually withdraw attacker-funded liquidity to free up bins before adding liquidity to new ones.

```
112 function fundLiquidityTraderJoeV2(ILBPair pair, uint256[] memory ids, uint256[] memory amounts)
        external nonReentrant {
113     if (!isPairWhitelisted(address(pair))) revert TraderJoeV2PoolNotWhitelisted();
114
115     pair.batchTransferFrom(msg.sender, address(this), ids, amounts);
116
117     TraderJoeV2Bin[] memory ownedBins = getOwnedTraderJoeV2Bins();
118
119     for (uint256 i; i < ids.length; ++i) {
120         bool userHadBin;
121
```

```
122         for (int256 j; uint(j) < ownedBins.length; ++j) {
123             if (address(ownedBins[uint(j)].pair) == address(pair)
124                 && ownedBins[uint(j)].id == ids[i]
125             ) {
126                 userHadBin = true;
127                 break;
128             }
129         }
130
131         if (!userHadBin) {
132             getOwnedTraderJoeV2BinsStorage().push(TraderJoeV2Bin(pair, uint24(ids[i])));
133         }
134     }
```

Listing 2.68: contracts/facets/TraderJoeV2Facet.sol

```
36  function getOwnedTraderJoeV2BinsStorage() internal returns (TraderJoeV2Bin[] storage result){
37      return DiamondStorageLib.getTjV2OwnedBins();
38  }
```

Listing 2.69: contracts/facets/TraderJoeV2Facet.sol

Impact A temporary DoS when adding liquidity.

Suggestion Apply access control to the `fundLiquidityTraderJoeV2` function.

2.2.26 Potential excessive deduction of protocol exposure

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `SmartLoanWrappedNativeTokenFacet` contract, the `unwrapAndWithdraw` function is used to unwrap the wrapped native token of a prime account and reduce its corresponding asset exposure within the protocol. However, since the protocol does not increase asset exposure for direct transfers of wrapped native tokens, malicious users can exploit the `unwrapAndWithdraw` function to reduce asset exposure based on their prime accounts' balance of wrapped native tokens.

Specifically, malicious users can first transfer wrapped native tokens directly to their prime accounts, then invoke the `unwrapAndWithdraw` function to maliciously decrease the exposure. Additionally, other instances where the `balanceOf` function is used for exposure adjustment, such as in the `withdraw` function of the `AssetsOperationsFacet` contract, may result in similar issues due to the lack of handling for direct transfers.

```
42  function unwrapAndWithdraw(uint256 _amount) onlyOwner remainsSolvent canRepayDebtFully public
        payable virtual {
43      IWrappedNativeToken wrapped = IWrappedNativeToken(DeploymentConstants.getNativeToken());
44      _amount = Math.min(wrapped.balanceOf(address(this)), _amount);
45      require(wrapped.balanceOf(address(this)) >= _amount, "Not enough native token to unwrap and
            withdraw");
46
```

```
47      wrapped.withdraw(_amount);
48
49      if (wrapped.balanceOf(address(this)) == 0) {
50          DiamondStorageLib.removeOwnedAsset(DeploymentConstants.getNativeTokenSymbol());
51      }
52
53      ITokenManager tokenManager = DeploymentConstants.getTokenManager();
54      _decreaseExposure(tokenManager, address(wrapped), _amount);
55
56      payable(msg.sender).safeTransferETH(_amount);
57
58      emit UnwrapAndWithdraw(msg.sender, _amount, block.timestamp);
59  }
```

Listing 2.70: contracts/facets/SmartLoanWrappedNativeTokenFacet.sol

```
116 function withdraw(bytes32 _withdrawnAsset, uint256 _amount) public virtual onlyOwner
        nonReentrant canRepayDebtFully remainsSolvent {
117     IERC20Metadata token = getERC20TokenInstance(_withdrawnAsset, true);
118     _amount = Math.min(_amount, token.balanceOf(address(this)));
119
120     address(token).safeTransfer(msg.sender, _amount);
121
122     ITokenManager tokenManager = DeploymentConstants.getTokenManager();
123
124     _decreaseExposure(tokenManager, address(token), _amount);
125     emit Withdrawn(msg.sender, _withdrawnAsset, _amount, block.timestamp);
126 }
```

Listing 2.71: contracts/facets/AssetsOperationFacet.sol

Impact Users can maliciously reduce exposure to whitelist tokens by performing direct transfers followed by withdrawals.

Suggestion Revise the code logic accordingly.

2.2.27 Unexpected operations by liquidator

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `OnlyOwnerOrInsolvent` contract, the modifiers `onlyOwnerOrInsolvent` and `onlyOwnerNoStaySolventOrInsolventPayable` allow liquidators to access users' prime accounts when they are insolvent and perform necessary liquidation operations. However, there is no restriction preventing liquidators from creating prime accounts. As a result, liquidators can perform operations to manipulate their insolvent positions (e.g., the `wrapNativeToken` function in the `SmartLoanWrappedNativeTokenFacet` contract).

```
19  modifier onlyOwnerOrInsolvent() {
20      bool isWhitelistedLiquidator = SmartLoanLiquidationFacet(DeploymentConstants.
            getDiamondAddress()).isLiquidatorWhitelisted(msg.sender);
```

```
21
22      if (isWhitelistedLiquidator) {
23          require(!_isSolvent(), "Account is solvent");
24      } else{
25          DiamondStorageLib.enforceIsContractOwner();
26      }
27
28      _;
29
30      if (!isWhitelistedLiquidator) {
31          require(_isSolvent(), "Must stay solvent");
32      }
33  }
34
35  modifier onlyOwnerNoStaySolventOrInsolventPayable() {
36      bool isWhitelistedLiquidator = SmartLoanLiquidationFacet(DeploymentConstants.
            getDiamondAddress()).isLiquidatorWhitelisted(msg.sender);
37
38      if (isWhitelistedLiquidator) {
39          require(!_isSolventPayable(), "Account is solvent");
40      } else{
41          DiamondStorageLib.enforceIsContractOwner();
42      }
43
44      _;
45  }
```

Listing 2.72: contracts/onlyOwnerOrInsolvent.sol

Impact Liquidators are allowed to manipulate their insolvent positions.

Suggestion Revise the code logic accordingly.

2.2.28 Unexpected revert due to dust token in the `liquidate` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `SmartLoanLiquidationFacet` contract, when the variable `healingLoan` is set to true, the return bonus in the `liquidate` function may cause precision loss, generating dust. This can lead to a failed check (line#250, which performs the calculation using `address(this).balance`) and cause the transaction to revert.

```
207  if(partToReturnBonus > 0){
208      // Native token transfer
209      if (address(this).balance > 0) {
210          uint256 transferAmount = address(this).balance * partToReturnBonus / 3 / 10 ** 18;
211          payable(DeploymentConstants.getStabilityPoolAddress()).safeTransferETH(transferAmount);
212          emit LiquidationTransfer(DeploymentConstants.getStabilityPoolAddress(),
                 DeploymentConstants.getNativeTokenSymbol(), transferAmount, block.timestamp);
213
214          payable(DeploymentConstants.getTreasuryAddress()).safeTransferETH(transferAmount);
```

```
215         emit LiquidationFeesTransfer(DeploymentConstants.getTreasuryAddress(),
                DeploymentConstants.getNativeTokenSymbol(), transferAmount, block.timestamp);
216
217         payable(DeploymentConstants.getFeesRedistributionAddress()).safeTransferETH(
                transferAmount);
218         emit LiquidationFeesRedistributionTransfer(DeploymentConstants.
                getFeesRedistributionAddress(), DeploymentConstants.getNativeTokenSymbol(),
                transferAmount, block.timestamp);
219     }
```

Listing 2.73: contracts/facets/SmartLoanLiquidationFacet.sol

```
248   if (healingLoan) {
249       require(_getDebtWithPrices(cachedPrices.debtAssetsPrices) == 0, "Healing a loan must end up
              with 0 debt");
250       require(_getTotalValueWithPrices(cachedPrices.ownedAssetsPrices, cachedPrices.
              stakedPositionsPrices) == 0, "Healing a loan must end up with 0 total value");
251   } else {
```

Listing 2.74: contracts/facets/SmartLoanLiquidationFacet.sol

Impact The dust token could lead to an unexpected revert during liquidation.

Suggestion Revise the code logic accordingly.

2.2.29 Inability to track real-time exposure

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description The `increaseProtocolExposure` and `decreaseProtocolExposure` functions of the `TokenManager` contract do not update `exposure.current` when `exposure.max` is zero for a given asset. If the owner resets `exposure.max` to a non-zero value via the `setMaxProtocolsExposure` function, this design could restore the previous asset exposure, causing `exposure.current` to misalign with the actual exposure throughout the protocol lifecycle.

```
118   function increaseProtocolExposure(bytes32 assetIdentifier, uint256 exposureIncrease) public
          onlyPrimeAccountOrOwner {
119       bytes32 group = identifierToExposureGroup[assetIdentifier];
120       if(group != ""){
121           Exposure storage exposure = groupToExposure[group];
122           if(exposure.max != 0){
123               exposure.current += exposureIncrease;
124               require(exposure.current <= exposure.max, "Max asset exposure breached");
125               emit ProtocolExposureChanged(msg.sender, group, exposure.current, block.timestamp);
126           }
127       }
128   }
129
130   function decreaseProtocolExposure(bytes32 assetIdentifier, uint256 exposureDecrease) public
          onlyPrimeAccountOrOwner {
131       bytes32 group = identifierToExposureGroup[assetIdentifier];
```

```
132       if(group != ""){
133           Exposure storage exposure = groupToExposure[group];
134           if(exposure.max != 0){
135               exposure.current = exposure.current <= exposureDecrease ? 0 : exposure.current -
                      exposureDecrease;
136               emit ProtocolExposureChanged(msg.sender, group, exposure.current, block.timestamp);
137           }
138       }
139   }
```

Listing 2.75: contracts/TokenManager.sol

Impact The protocol cannot record real-time asset exposures when `exposure.max` is reset from zero to a non-zero value.

Suggestion Revise the code logic accordingly.

2.2.30 Inconsistent health factor calculation between contracts `SolvencyFacetProd` and `HealthMeterFacetProd`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `SolvencyFacetProd` contract, the health ratio calculation includes Uniswap-V3 positions, whereas the `HealthMeterFacetProd` contract does not account for them in its health meter calculation. This inconsistency could impact certain off-chain operations, such as the initialization of liquidations.

```
333   function _getThresholdWeightedValueBase(AssetPrice[] memory ownedAssetsPrices, AssetPrice[]
          memory stakedPositionsPrices) internal view virtual returns (uint256) {
334       return _getTWVOwnedAssets(ownedAssetsPrices) + _getTWVStakedPositions(stakedPositionsPrices
              ) + _getTotalTraderJoeV2(true) + _getTotalUniswapV3(true);
335   }
```

Listing 2.76: contracts/facets/SolvencyFacetProd.sol

```
233   function getHealthMeter() public view returns (uint256) {
234       AssetPrice[] memory assetsPrices = _getAllAssetsWithNativePrices();
235
236       bytes32 nativeTokenSymbol = DeploymentConstants.getNativeTokenSymbol();
237       ITokenManager tokenManager = DeploymentConstants.getTokenManager();
238
239       uint256 weightedCollateralPlus = assetsPrices[0].price * address(this).balance *
              tokenManager.debtCoverage(tokenManager.getAssetAddress(nativeTokenSymbol, true)) / (10
              ** 26);
240       uint256 weightedCollateralMinus = 0;
241       uint256 weightedBorrowed = 0;
242       uint256 borrowed = 0;
243
244       weightedCollateralPlus += _getTotalTraderJoeV2Weighted();
245       weightedCollateralPlus += _getTWVStakedPositions();
```

```
246
247       for (uint256 i = 0; i < assetsPrices.length; i++) {
```

Listing 2.77: contracts/facets/HealthMeterFacetProd.sol

Impact The inconsistent health ratio calculation may impact off-chain operations.

Suggestion Remove the unused calculation and adjust the code logic accordingly.

2.2.31 Improper modifier usage in the `swapDebtParaSwap` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `AssetsOperationsFacet` contract, the function `swapDebtParaSwap` uses the modifier `onlyOwnerOrInsolvent`, allowing owners and whitelisted liquidators to swap users' debt. This is inconsistent with the `swapDebt` function, which performs similar logic with a different exchange (i.e., Yak) but only allows invocation by the prime account owner.

Moreover, the `swapDebtParaSwap` function fails to handle unused `_toAsset` amounts. Unlike `swapDebt`, which precisely controls the swap amount by using the full borrowed amount as `amountIn`, `swapDebtParaSwap` uses caller-specified `data` as swap parameters. If a user does not already own `_toAsset`, it is not included in the user's `ownedAssets` mapping. As a result, if the swap does not consume the entire borrowed `_toAsset`, the remaining amount is excluded from the asset valuation, causing the health ratio to be lower than it should be after the debt swap.

```
233   function swapDebt(bytes32 _fromAsset, bytes32 _toAsset, uint256 _repayAmount, uint256
          _borrowAmount, address[] calldata _path, address[] calldata _adapters) external onlyOwner
          remainsSolvent nonReentrant {
```

Listing 2.78: contracts/facets/AssetsOperationsFacet.sol

```
271   function swapDebtParaSwap(bytes32 _fromAsset, bytes32 _toAsset, uint256 _repayAmount, uint256
          _borrowAmount, bytes4 selector, bytes memory data) external onlyOwnerOrInsolvent
          remainsSolvent nonReentrant {
272       ITokenManager tokenManager = DeploymentConstants.getTokenManager();
273       Pool fromAssetPool = Pool(tokenManager.getPoolAddress(_fromAsset));
274       _repayAmount = Math.min(_repayAmount, fromAssetPool.getBorrowed(address(this)));
275
276       IERC20Metadata toToken = getERC20TokenInstance(_toAsset, false);
277       IERC20Metadata fromToken = getERC20TokenInstance(_fromAsset, false);
278
279       Pool toAssetPool = Pool(tokenManager.getPoolAddress(_toAsset));
280       toAssetPool.borrow(_borrowAmount);
281
282       uint256 initialRepayTokenAmount = fromToken.balanceOf(address(this));
283
284       {
285
286           // swap toAsset to fromAsset
287           address(toToken).safeApprove(PARA_TRANSFER_PROXY, 0);
```

```
288             address(toToken).safeApprove(PARA_TRANSFER_PROXY, _borrowAmount);
289
290             (bool success, ) = PARA_ROUTER.call((abi.encodePacked(selector, data)));
291             require(success, "Swap failed");
292
293         }
294         _repayAmount = Math.min(fromToken.balanceOf(address(this)), _repayAmount);
295
296         _processRepay(tokenManager, fromAssetPool, address(fromToken), _repayAmount, fromToken.
                balanceOf(address(this)) - initialRepayTokenAmount);
297
298         emit DebtSwap(msg.sender, address(fromToken), address(toToken), _repayAmount, _borrowAmount
                , block.timestamp);
299     }
```

Listing 2.79: contracts/facets/AssetsOperationsFacet.sol

Impact Liquidators could swap debts on behalf of prime account owners, causing unexpected consequences.

Suggestion Change the modifier from `onlyOwnerOrInsolvent` to `onlyOwner`.

2.2.32 Lack of validation for duplicate users in `eligibleUsersList`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `BtcEligibleUsersList` contract, the function `addEligibleUsers` lacks validation for duplicate users. As a result, a user may appear multiple times in the `eligibleUsersList`.

```
15  function addEligibleUsers(address[] calldata _eligibleUsers) external onlyOwner {
16      for (uint256 i = 0; i < _eligibleUsers.length; i++) {
17          eligibleUsersList.push(_eligibleUsers[i]);
18      }
19  }
20
21  /// only owner function to remove users from eligible list
22  function removeEligibleUsers(address[] calldata _eligibleUsers) external onlyOwner {
23      for (uint256 i = 0; i < _eligibleUsers.length; i++) {
24          for (uint256 j = 0; j < eligibleUsersList.length; j++) {
25              if (eligibleUsersList[j] == _eligibleUsers[i]) {
26                  eligibleUsersList[j] = eligibleUsersList[eligibleUsersList.length - 1];
27                  eligibleUsersList.pop();
28              }
29          }
30      }
31  }
```

Listing 2.80: contracts/BtcEligibleUsersList.sol

This lack of validation could lead to the following potential issues:

- The `removeEligibleUsers` function may not fully remove a user from the list. Here are two examples:
  - Case 1 (normal):
    - Add [a, b, c] → [a, b, c]
    - Add [a, b, c] → [a, b, c, a, b, c]
    - Remove [a] → [b, c, b, c]
  - Case 2 (problematic):
    - Add [a, b, c] → [a, b, c]
    - Add [a] → [a, b, c, a]
    - Remove [a] → [a, b, c]
- The `fulfillRandomWords` function in the `RandomTokenRewarder` contract rewards a randomly selected user from the `eligibleUsersList` in the `BtcEligibleUsersList` contract. A user who appears multiple times in the list would have a higher chance of winning the reward.

```
69   function fulfillRandomWords(
70       uint256 _requestId,
71       uint256[] memory _randomWords
72   ) internal override {
73       uint256 randomWord = _randomWords[0];
74
75       uint256 primeAccountCount = btcEligibleUsersList.getEligibleUsersCount();
76       address primeAccountAddress = btcEligibleUsersList.eligibleUsersList(randomWord %
             primeAccountCount);
77
78       uint256 rewardBalance = IERC20(rewardToken).balanceOf(address(this));
79       IERC20(rewardToken).transfer(primeAccountAddress, rewardBalance);
80       emit RewardTransferred(primeAccountAddress, rewardToken, rewardBalance, block.number);
81   }
```

Listing 2.81: contracts/RandomTokenRewarder.sol

Impact The lack of a duplication check in `eligibleUsersList` could result in incomplete removal of duplicated users or cause unfairness during the winner determination process.

Suggestion Add a duplication check for `eligibleUsersList`.

Feedback from the project The project stated that the `BtcEligibleUsersList` contract was intended for one-time use, and they have since removed it.

2.2.33 Lack of a reentrancy guard in the `createAndFundLoan` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description The `createAndFundLoan` function in the `SmartLoansFactory` contract is vulnerable to reentrancy attacks. If the funded ERC20 `token` (e.g., ERC777, ERC721, ERC1155) implements a callback mechanism, attackers could re-enter the contract during the token's transfer logic (Line#95), leading to unexpected consequences.

```
86   function createAndFundLoan(bytes32 _fundedAsset, uint256 _amount) public virtual hasNoLoan
         returns (SmartLoanDiamondBeacon) {
87       address asset = tokenManager.getAssetAddress(_fundedAsset, false);
88       SmartLoanDiamondProxy beaconProxy = new SmartLoanDiamondProxy(payable(address(
             smartLoanDiamond)),
89           abi.encodeWithSelector(SmartLoanViewFacet.initialize.selector, msg.sender)
90       );
91       SmartLoanDiamondBeacon smartLoan = SmartLoanDiamondBeacon(payable(address(beaconProxy)));
92
93       //Fund account with own funds and credit
94       IERC20Metadata token = IERC20Metadata(asset);
95       address(token).safeTransferFrom(msg.sender, address(this), _amount);
96       address(token).safeApprove(address(smartLoan), _amount);
97
98       //Update registry and emit event
99       updateRegistry(address(smartLoan), msg.sender);
100
101      (bool success, bytes memory result) = address(smartLoan).call(abi.encodeWithSelector(
             AssetsOperationsFacet.fund.selector, _fundedAsset, _amount));
102      ProxyConnector._prepareReturnValue(success, result);
103
104      emit SmartLoanCreated(address(smartLoan), msg.sender, _fundedAsset, _amount);
105
106      return smartLoan;
107  }
```

Listing 2.82: contracts/SmartLoansFactory.sol

Impact Attackers could initiate a reentrancy attack through the `createAndFundLoan` function.

Suggestion Add the `nonReentrant` modifier to the function.

2.2.34 Lack of validation for the parameter `_adapters` in the `yakSwap` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `YieldYakSwapFacet` contract, the `yakSwap` function accepts a user-specified parameter `_adapters` and passes it to the Yak router (`YY_ROUTER`) to perform swaps. However, there is no validation of the input `_adapters`, which allows an attacker to pass a malicious adapter that could trigger unexpected behavior.

Specifically, an attacker could deploy a contract with customized functions (e.g., `swap` and `query`, which are invoked in `YY_ROUTER`) to carry out malicious actions. Although the `yakSwap` function includes the `remainsSolvent` modifier, this lack of input validation could still lead to security risks.

- **Case 1 (reentrancy):** An attacker could leverage the above vulnerability to reenter their prime account to manipulate the variable `boughtTokenFinalAmount` without conducting swaps. Here is a detailed example:

1. The attacker deploys a malicious adapter contract with customized `query` and `swap` functions, which include reentrancy logic.
2. The attacker invokes the `refundExecutionFee` function within the `swap` function of the adapter contract.
3. The attacker then invokes the `yakSwap` function with following inputs:
   - A `_path` variable, setting `boughtToken` as the wrapped native token.
   - A `_adapters` variable, which includes the malicious adapter contract.
4. Since `refundExecutionFee` wraps all native tokens, the `boughtTokenFinalAmount` variable is manipulated without any swap occurring.
5. As a result, the invocation of the `_increaseExposure` function increases the exposure of the wrapped native token.

- **Case 2 (violation of the withdrawal guard design):** Moreover, with the customized adapter contract, an attacker could withdraw assets from their prime account using the `yakSwap` function. During the execution of the `swapNoSplit` function in the `YY_ROUTER` contract, the asset is first transferred from the prime account to the adapter contract. If the attacker implements a transfer within the adapter contract, the received assets could be sent directly to EOAs, enabling the attacker to withdraw assets from the prime account without fully holding the associated debt.

```
59  function yakSwap(uint256 _amountIn, uint256 _amountOut, address[] calldata _path, address[]
        calldata _adapters) external nonReentrant onlyOwner noBorrowInTheSameBlock remainsSolvent{
60      SwapTokensDetails memory swapTokensDetails = getInitialTokensDetails(_path[0], _path[_path.
            length - 1]);
61
62      _amountIn = Math.min(swapTokensDetails.soldToken.balanceOf(address(this)), _amountIn);
63      require(_amountIn > 0, "Amount of tokens to sell has to be greater than 0");
64
65      address(swapTokensDetails.soldToken).safeApprove(YY_ROUTER(), 0);
66      address(swapTokensDetails.soldToken).safeApprove(YY_ROUTER(), _amountIn);
67
68      IYieldYakRouter router = IYieldYakRouter(YY_ROUTER());
69
70      IYieldYakRouter.Trade memory trade = IYieldYakRouter.Trade({
71          amountIn: _amountIn,
72          amountOut: _amountOut,
73          path: _path,
74          adapters: _adapters
75      });
76
77      router.swapNoSplit(trade, address(this), 0);
```

Listing 2.83: contracts/facets/avalanche/YieldYakSwapFacet.sol

```
551  function _swapNoSplit(
552      Trade calldata _trade,
553      address _from,
554      address _to,
555      uint _fee
556  ) internal returns (uint) {
557      // ...
```

```
558     for (uint256 i=0; i<_trade.adapters.length; i++) {
559         // All adapters should transfer output token to the following target
560         // All targets are the adapters, expect for the last swap where tokens are sent out
561         address targetAddress = i<_trade.adapters.length-1 ? _trade.adapters[i+1] : _to;
562         IAdapter(_trade.adapters[i]).swap(
563             amounts[i],
564             amounts[i+1],
565             _trade.path[i],
566             _trade.path[i+1],
567             targetAddress
568         );
569     }
```

Listing 2.84: YakRouter.sol on Avalanche chain

Impact Attackers could maliciously increase the protocol's exposure or withdraw their assets from their prime accounts without full debt.

Suggestion Revise the code logic accordingly.

2.2.35 Lack of validation for additional rewards received

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description The `WombatFacet` contract lacks validation to ensure the prime account receives the correct reward amount from `IWombatMaster.withdraw`, as specified by the `additionalRewards` return value. It directly passes the `additionalRewards` array to the `handleRewards` function, which either adds the array's elements to the current exposure or transfers corresponding rewards to the account owner. This approach introduces risks, as the returned value from Wombat may be counterfeit if any pool rewarder is malicious or compromised. Relying solely on invalidated return values for handling rewards could lead to unexpected consequences. The same issue exists in the `YieldYakWombatFacet` contract.

```
296   function claimAllWombatRewards()
297       external
298       onlyOwner
299       nonReentrant
300       remainsSolvent
301   {
302       bytes32[4] memory lpAssets = [
303           WOMBAT_ggAVAX_AVAX_LP_AVAX,
304           WOMBAT_ggAVAX_AVAX_LP_ggAVAX,
305           WOMBAT_sAVAX_AVAX_LP_AVAX,
306           WOMBAT_sAVAX_AVAX_LP_sAVAX
307       ];
308       for (uint256 i; i != 4; ++i) {
309           IERC20Metadata lpToken = getERC20TokenInstance(lpAssets[i], false);
310           uint256 pid = IWombatMaster(WOMBAT_MASTER).getAssetPid(address(lpToken));
311           (uint256 reward, uint256[] memory additionalRewards) = IWombatMaster(
```

```
312             WOMBAT_MASTER
313         ).withdraw(pid, 0);
314         handleRewards(pid, reward, additionalRewards);
315     }
316 }
```

Listing 2.85: contracts/facets/avalanche/WombatFacet.sol

```
642 function handleRewards(
643     uint256 pid,
644     uint256 reward,
645     uint256[] memory additionalRewards
646 ) internal {
647     // ...
648
649     uint256 baseIdx;
650     if (rewarder != address(0)) {
651         address[] memory rewardTokens = IRewarder(rewarder).rewardTokens();
652         baseIdx = rewardTokens.length;
653         for (uint256 i; i != baseIdx; ++i) {
654             address rewardToken = rewardTokens[i];
655             uint256 pendingReward = additionalRewards[i];
656
657             if (pendingReward == 0) {
658                 continue;
659             }
660
661             if (tokenManager.isTokenAssetActive(rewardToken)) {
662                 _increaseExposure(tokenManager, rewardToken, pendingReward);
663             } else {
664                 rewardToken.safeTransfer(owner, pendingReward);
665             }
666         }
667     }
668     if (boostedRewarder != address(0)) {
669         address[] memory rewardTokens = IRewarder(boostedRewarder).rewardTokens();
670         for (uint256 i; i != rewardTokens.length; ++i) {
671             address rewardToken = rewardTokens[i];
672             uint256 pendingReward = additionalRewards[baseIdx + i];
673
674             if (pendingReward == 0) {
675                 continue;
676             }
677
678             if (tokenManager.isTokenAssetActive(rewardToken)) {
679                 _increaseExposure(tokenManager, rewardToken, pendingReward);
680             } else {
681                 rewardToken.safeTransfer(owner, pendingReward);
682             }
683         }
684     }
685 }
```

Listing 2.86: contracts/facets/avalanche/WombatFacet.sol

Impact Unverified `additionalRewards` amounts may lead to exposure manipulation or unexpected token transfers.

Suggestion Implement validation logic for the arrival of additional rewards.

2.2.36 Improper condition check in the `createWithdrawalIntent` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-2`

Description In the `Pool` contract, the `createWithdrawalIntent` function checks the `amount` and `msg.sender`'s available balance. However, the check on line#270 is problematic because `totalIntentAmount` is already included in the return value of `getNotLockedBalance(msg.sender, 0)`. As a result, when a user creates a withdrawal intent exceeding 50% of their available balance, the withdrawal will always fail due to the repeated accounting of `totalIntentAmount`.

```
261   function createWithdrawalIntent(uint256 amount) external {
262       require(amount > 0, "Amount must be greater than zero");
263
264       // Remove expired intents first
265       _removeExpiredIntents(msg.sender);
266
267       uint256 totalIntentAmount = getTotalIntentAmount(msg.sender);
268       uint256 availableBalance = getNotLockedBalance(msg.sender, 0);
269
270       require(amount + totalIntentAmount <= availableBalance, "Insufficient available balance");
271
272       uint256 actionableAt = block.timestamp + 24 hours;
273       uint256 expiresAt = actionableAt + 24 hours;
274
275       WithdrawalIntent memory newIntent = WithdrawalIntent({
276           amount: amount,
277           actionableAt: actionableAt,
278           expiresAt: expiresAt
279       });
280
281       withdrawalIntents[msg.sender].push(newIntent);
282
283       emit WithdrawalIntentCreated(msg.sender, amount, actionableAt, expiresAt);
284   }
```

Listing 2.87: contracts/Pool.sol

```
92    function getNotLockedBalance(address account, uint256 excludedIntentAmount) public view returns
          (uint256 notLockedBalance) {
93        uint256 lockedBalance = getLockedBalance(account);
94        uint256 totalIntentAmount = getTotalIntentAmount(account);
95
96        // Subtract the excluded intent amount
97        totalIntentAmount = totalIntentAmount > excludedIntentAmount ? totalIntentAmount -
              excludedIntentAmount : 0;
```

```
98
99      uint256 balance = balanceOf(account);
100     uint256 unavailableBalance = lockedBalance + totalIntentAmount;
101
102     if (balance < unavailableBalance) {
103         notLockedBalance = 0;
104     } else {
105         notLockedBalance = balance - unavailableBalance;
106     }
107 }
```

Listing 2.88: contracts/Pool.sol

Impact May result in a potential DoS when creating withdrawal intentions.

Suggestion Revise the code logic accordingly.

2.2.37 Reusable withdrawal intents due to reentrancy issue

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-2`

Description The `withdrawNativeToken` function has a reentrancy issue that allows users to reuse withdrawal intents multiple times. Specifically, native tokens are transferred through `payable(msg.sender).safeTransferETH(_amount)` before updating the `intents` array. This transfer allows malicious actors to reenter the contract through the `fallback` function. Exploiting this, the `fallback` function can invoke `cancelWithdrawalIntent` and `createWithdrawalIntent`, preventing the removal of the active withdrawal intent. As a result, the preserved intent can be reused for withdrawals as long as the user holds sufficient pool tokens.

Specifically, the attacker could take the following steps to launch a example attack:

1. The attacker starts with three active withdrawal intents: *[1, 1, 100]*.
2. The attacker calls `withdrawNativeToken(100, 2)` and reenters the contract in the `fallback` function:
   (a). The attacker calls `cancelWithdrawalIntent(1)`, updating the withdrawal intents to: *[1, 100]*.
   (b). The attacker then calls `createWthdrawalIntent(10)`, updating the withdrawal intents to: *[1, 100, 10]*.
3. During the removal of withdrawal intents, the last element (the newly created intent from Step 2.b) is removed, leaving *[1, 100]*. Consequently, the previously used intent with an amount of 100 remains in the array and can be reused.

The same issue exists in the `withdraw` function of the `Pool` contract if the withdrawn ERC20 token supports a callback mechanism.

```
50  function withdrawNativeToken(uint256 _amount, uint256 intentIndex) external nonReentrant {
51      WithdrawalIntent[] storage intents = withdrawalIntents[msg.sender];
52
53      // ...
54
```

```
55      // Transfer the native tokens to the user
56      payable(msg.sender).safeTransferETH(_amount);
57
58      // ...
59
60      // Remove the used intent
61      uint256 lastIndex = intents.length - 1;
62      if (intentIndex != lastIndex) {
63          intents[intentIndex] = intents[lastIndex];
64      }
65      intents.pop();
66  }
```

Listing 2.89: contracts/WrappedNativeTokenPool.sol

```
499  function withdraw(uint256 _amount, uint256 intentIndex) external nonReentrant {
500      WithdrawalIntent[] storage intents = withdrawalIntents[msg.sender];
501      // ...
502
503      _transferFromPool(msg.sender, _amount);
504
505      // ...
506
507      // Remove the used intent
508      uint256 lastIndex = intents.length - 1;
509      if (intentIndex != lastIndex) {
510          intents[intentIndex] = intents[lastIndex];
511      }
512      intents.pop();
```

Listing 2.90: contracts/Pool.sol

Impact The flawed design allows malicious users to reuse existing withdrawal intents.

Suggestion Apply a reentrancy guard to the vulnerable functions.

2.2.38 Incomplete withdrawal due to exclusion of accrued interest

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1-2`

Description In the `Pool` and `WrappedNativeTokenPool` contracts, the withdrawal lock mechanism allows users to create a withdrawal intent with a specified amount and withdraw their assets after 24 hours. The specified amount must not exceed the available unlocked balance. However, the withdrawal intent creation does not account for interest accrued during the 24-hour waiting period. Additionally, the `withdraw` function includes a `require` check ensuring that the withdrawal amount (i.e., `_amount`) matches the proposed intent amount (line#504). As a result, when users attempt to withdraw all available assets, any accrued interest remains in the pool, resulting in incomplete withdrawals.

```
261   function createWithdrawalIntent(uint256 amount) external {
262       require(amount > 0, "Amount must be greater than zero");
263
264       // Remove expired intents first
265       _removeExpiredIntents(msg.sender);
266
267       uint256 totalIntentAmount = getTotalIntentAmount(msg.sender);
268       uint256 availableBalance = getNotLockedBalance(msg.sender, 0);
269
270       require(amount + totalIntentAmount <= availableBalance, "Insufficient available balance");
271
272       uint256 actionableAt = block.timestamp + 24 hours;
273       uint256 expiresAt = actionableAt + 24 hours;
274
275       WithdrawalIntent memory newIntent = WithdrawalIntent({
276           amount: amount,
277           actionableAt: actionableAt,
278           expiresAt: expiresAt
279       });
280
281       withdrawalIntents[msg.sender].push(newIntent);
282
283       emit WithdrawalIntentCreated(msg.sender, amount, actionableAt, expiresAt);
284   }
```

Listing 2.91: contracts/Pool.sol

```
499   function withdraw(uint256 _amount, uint256 intentIndex) external nonReentrant {
500       WithdrawalIntent[] storage intents = withdrawalIntents[msg.sender];
501       require(intentIndex < intents.length, "Invalid intent index");
502
503       WithdrawalIntent storage intent = intents[intentIndex];
504       require(intent.amount == _amount, "Withdrawal amount must match intent amount");
```

Listing 2.92: contracts/Pool.sol

Impact Users cannot perform a complete withdrawal that includes accrued interest.

Suggestion Revise the code logic accordingly.

2.3 Additional Recommendation

2.3.1 Remove the unused `prevIndex` assignment in the `updateIndex` function

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the function `updateIndex`, the assignment to `prevIndex` for `indexUpdateTime` (line#93) is unnecessary and can be removed. Since `indexUpdateTime` represents the global rate update timestamp and `prevIndex[indexUpdateTime]` is never read, tracking `prevIndex` in `updateIndex` is redundant. Only the tracking in the `updateUser` function is used.

```
92  function updateIndex() internal {
93      prevIndex[indexUpdateTime] = index;
94
95      index = getIndex();
96      indexUpdateTime = block.timestamp;
97  }
```

Listing 2.93: contracts/LinearIndex.sol

```
44  function setRate(uint256 _rate) public override onlyOwner {
45      updateIndex();
46      rate = _rate;
47      emit RateUpdated(rate, block.timestamp);
48  }
```

Listing 2.94: contracts/LinearIndex.sol

```
57  function updateUser(address user) public override onlyOwner {
58      userUpdateTime[user] = block.timestamp;
59      prevIndex[block.timestamp] = getIndex();
60  }
```

Listing 2.95: contracts/LinearIndex.sol

```
83  function getIndexedValue(uint256 value, address user) public view override returns (uint256) {
84      uint256 userTime = userUpdateTime[user];
85      uint256 prevUserIndex = userTime == 0 ? getIndex() : prevIndex[userTime];
86
87      return value * getIndex() / prevUserIndex;
88  }
```

Listing 2.96: contracts/LinearIndex.sol

Impact May lead to unnecessary gas usage and create ambiguity in the code's logic.

Suggestion Remove the unnecessary assignment in the `updateIndex` function.

2.3.2 Remove the unused state variable `_status` in the `ReentrancyGuardKeccak` contract

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `ReentrancyGuardKeccak`, the private variable `_status` is defined and assigned in the constructor. However, the `nonReentrant` modifier checks for non-reentrancy using the variable `rgs._status`, which is retrieved from `reentrancyGuardStorage` of the contract `DiamondStorageLib`, leading to potential inconsistency in the reentrancy status tracking.

```
20  uint256 private _status;
21
22  constructor() {
23      _status = _NOT_ENTERED;
```

```
24  }
```

Listing 2.97: contracts/ReentrancyGuardKeccak.sol

```
33  modifier nonReentrant() {
34      DiamondStorageLib.ReentrancyGuardStorage storage rgs = DiamondStorageLib.
            reentrancyGuardStorage();
35      // On the first call to nonReentrant, _notEntered will be true
36      require(rgs._status != _ENTERED, "ReentrancyGuard: reentrant call");
37
38      // Any calls to nonReentrant after this point will fail
39      rgs._status = _ENTERED;
40
41      _;
42
43      // By storing the original value once again, a refund is triggered (see
44      // https://eips.ethereum.org/EIPS/eip-2200)
45      rgs._status = _NOT_ENTERED;
46  }
```

Listing 2.98: contracts/ReentrancyGuardKeccak.sol

Impact May lead to unexpected results.

Suggestion Remove the unused variable `_status` in the `ReentrancyGuardKeccak` contract.

2.3.3 Remove the redundant validation in the `unwrapAndWithdraw` function

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `SmartLoanWrappedNativeTokenFacet`, the first `require` check of the function `unwrapAndWithdraw` (line#45) is redundant, as the value of `_amount` has already been set to the minimum between `wrapped.balanceOf(address(this))` and the original `_amount` on line#44.

```
42  function unwrapAndWithdraw(uint256 _amount) onlyOwner remainsSolvent canRepayDebtFully public
        payable virtual {
43      IWrappedNativeToken wrapped = IWrappedNativeToken(DeploymentConstants.getNativeToken());
44      _amount = Math.min(wrapped.balanceOf(address(this)), _amount);
45      require(wrapped.balanceOf(address(this)) >= _amount, "Not enough native token to unwrap and
            withdraw");
```

Listing 2.99: contracts/facets/SmartLoanWrappedNativeTokenFacet.sol

Impact Redundant checks result in unnecessary gas consumption.

Suggestion Remove the redundant validation in the `unwrapAndWithdraw` function.

2.3.4 Remove the redundant import in the `AssetsOperationsAvalancheFacet` contract

Status Fixed in `Version 2`

Introduced by Version 1-1

Description Compared to the contract AssetsOperationsArbitrumFacet, the import of *"../SmartLonLiquidationFacet.sol"* in AssetsOperationsAvalancheFacet is redundant.

```
5 import "../AssetsOperationsFacet.sol";
6 import "../SmartLoanLiquidationFacet.sol";
7
8 contract AssetsOperationsAvalancheFacet is AssetsOperationsFacet {
```

Listing 2.100: contracts/facets/avalanche/AssetsOperationsAvalancheFacet.sol

Impact N/A

Suggestion Remove the redundant import.

2.3.5 Remove the redundant validation in the functions _withdrawFromPool and _depositToPool

Status Fixed in Version 2

Introduced by Version 1-1

Description In the function _withdrawFromPool of the DepositSwap contract, the first require check (line#27) is unnecessary. The input amount is already constrained to the minimum value between pool.balanceOf(user) and the original amount in the external functions depositSwap and depositSwapParaSwap.

```
23   function _withdrawFromPool(Pool pool, IERC20 token, uint256 amount, address user) internal {
24       uint256 userInitialFromTokenDepositBalance = pool.balanceOf(user);
25       uint256 poolInitialBalance = pool.balanceOf(address(this));
26
27       require(userInitialFromTokenDepositBalance >= amount, "Insufficient fromToken deposit
             balance");
```

Listing 2.101: contracts/DepositSwap.sol

```
80   function depositSwap(uint256 amountFromToken, uint256 minAmountToToken, address[] calldata path
         , address[] calldata adapters) public {
81       // ...
82
83       Pool fromPool = _tokenToPoolTUPMapping(fromToken);
84       Pool toPool = _tokenToPoolTUPMapping(toToken);
85
86       address user = msg.sender;
87       amountFromToken = Math.min(fromPool.balanceOf(user), amountFromToken);
88
89       _withdrawFromPool(fromPool, IERC20(fromToken), amountFromToken, user);
```

Listing 2.102: contracts/DepositSwap.sol

```
101  function depositSwapParaSwap(
102      bytes4 selector,
103      bytes memory data,
104      address fromToken,
```

```
105         uint256 fromAmount,
106         address toToken,
107         uint256 minOut
108     ) public {
109         // ...
110
111         Pool fromPool = _tokenToPoolTUPMapping(fromToken);
112         Pool toPool = _tokenToPoolTUPMapping(toToken);
113
114         address user = msg.sender;
115         fromAmount = Math.min(fromPool.balanceOf(user), fromAmount);
116
117         _withdrawFromPool(fromPool, IERC20(fromToken), fromAmount, user);
118
119         // ...
120
121         uint256 amountOut = IERC20(toToken).balanceOf(address(this));
122         require(amountOut >= minOut, "Too little received");
123
124         _depositToPool(toPool, IERC20(toToken), amountOut, user);
125     }
```

Listing 2.103: contracts/DepositSwap.sol

Impact Redundant checks lead to unnecessary gas consumption.

Suggestion Remove the redundant checks in the function `_withdrawFromPool`.

2.3.6 Ensure balance check before removing pool assets

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `removePoolAssets` function of the `TokenManager` contract, pools are removed without verifying whether they are empty (i.e., whether all assets have been withdrawn and all debts repaid). It is recommended to implement a balance check to ensure that all assets are withdrawn and all debts are cleared before removing pools from the list.

```
267     function removePoolAssets(bytes32[] memory _poolAssets) public onlyOwner {
268         for (uint256 i = 0; i < _poolAssets.length; i++) {
269             _removePoolAsset(_poolAssets[i]);
270         }
271     }
272
273     function _removePoolAsset(bytes32 _poolAsset) internal {
274         address poolAddress = getPoolAddress(_poolAsset);
275         EnumerableMap.remove(assetToPoolAddress, _poolAsset);
276         emit PoolAssetRemoved(msg.sender, _poolAsset, poolAddress, block.timestamp);
277     }
```

Listing 2.104: contracts/TokenManager.sol

Impact This could cause unexpected losses.

Suggestion Add a validation check to ensure that pools are removed only when they are empty.

2.3.7 Implement a validation check for the function's parameters

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description The `addOwnedAsset` function allows any liquidator to add an asset to a prime account's `ownedAssets` mapping. However, it does not verify whether the `_asset` matches its corresponding `_address` in the `TokenManager` configuration. If a liquidator mistakenly provides mismatched parameters, the function may update the user's `ownedAssets` mapping incorrectly, potentially causing a miscalculation of the account's total value.

```
73   function addOwnedAsset(bytes32 _asset, address _address) external onlyWhitelistedLiquidators
         nonReentrant{
74       ITokenManager tokenManager = DeploymentConstants.getTokenManager();
75       require(tokenManager.isTokenAssetActive(_address), "Asset not supported");
76
77       DiamondStorageLib.addOwnedAsset(_asset, _address);
78   }
```

Listing 2.105: contracts/facets/AssetsOperationsFacet.sol

Impact The misalignment could lead to an improper value calculation for accounts.

Suggestion Add a validation check for the input parameters of the `addOwnedAsset` function.

2.3.8 Remove the redundant `payable` modifier

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `SmartLoanLiquidationFacet` contract, the `liquidateLoan` function is marked as `payable`. However, since the repayment value calculation only involves ERC20 tokens, the `payable` modifier appears redundant.

```
115   function liquidateLoan(bytes32[] memory assetsToRepay, uint256[] memory amountsToRepay, uint256
          _liquidationBonusPercent) external payable onlyWhitelistedLiquidators accountNotFrozen
          nonReentrant {
```

Listing 2.106: contracts/facets/SmartLoanLiquidationFacet.sol

Impact N/A

Suggestion Remove the redundant modifier `payable`.

2.3.9 Add a validation check to prevent the executing meaningless transactions

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `Timelock` contract, the `queueTransaction` function allows the admin to add pending transactions, consisting of `target`, `value`, `signature`, `data`, and `eta`. Then, in the function `executeTransaction`, these pending transactions are executed using a `call` instruction (i.e., `target.callvalue: value(callData)`). It is recommended to add a zero-address validation for the `target` variable in the function `queueTransaction` to prevent the execution of such meaningless transactions.

```
62  function queueTransaction(address target, uint value, string memory signature, bytes memory
        data, uint eta) public returns (bytes32) {
63      require(msg.sender == admin, "Timelock::queueTransaction: Call must come from admin.");
64      require(eta >= getBlockTimestamp().add(delay), "Timelock::queueTransaction: Estimated
            execution block must satisfy delay.");
65
66      bytes32 txHash = keccak256(abi.encode(target, value, signature, data, eta));
67      queuedTransactions[txHash] = true;
```

Listing 2.107: contracts/TimeLock.sol

```
82  function executeTransaction(address target, uint value, string memory signature, bytes memory
        data, uint eta) public payable returns (bytes memory) {
83      require(msg.sender == admin, "Timelock::executeTransaction: Call must come from admin.");
84
85      bytes32 txHash = keccak256(abi.encode(target, value, signature, data, eta));
86      require(queuedTransactions[txHash], "Timelock::executeTransaction: Transaction hasn't been
            queued.");
87      require(getBlockTimestamp() >= eta, "Timelock::executeTransaction: Transaction hasn't
            surpassed time lock.");
88      require(getBlockTimestamp() <= eta.add(GRACE_PERIOD), "Timelock::executeTransaction:
            Transaction is stale.");
89
90      queuedTransactions[txHash] = false;
91
92      bytes memory callData;
93
94      if (bytes(signature).length == 0) {
95          callData = data;
96      } else {
97          callData = abi.encodePacked(bytes4(keccak256(bytes(signature))), data);
98      }
99
100     // solium-disable-next-line security/no-call-value
101     (bool success, bytes memory returnData) = target.call{value: value}(callData);
102     require(success, "Timelock::executeTransaction: Transaction execution reverted.");
```

Listing 2.108: contracts/TimeLock.sol

Impact This could result in the execution of multiple meaningless transactions where the target is the zero address.

Suggestion Add a zero-address validation in this function.

2.3.10 Unify the handling of `msg.value` in the `_getTWVOwnedAssets` function across different chains

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `SolvencyFacetProdArbitrum` contract, the function `_getTWVOwnedAssets` calculates the value of owned assets by excluding `msg.value`. In contrast, the same function in the `SolvencyFacetProdAvalanche` contract does not perform this deduction. This discrepancy in handling `msg.value` could lead to inconsistent asset valuations across different chains.

```
62  function _getTWVOwnedAssets(AssetPrice[] memory ownedAssetsPrices) internal virtual override
        view returns (uint256) {
63      bytes32 nativeTokenSymbol = DeploymentConstants.getNativeTokenSymbol();
64      ITokenManager tokenManager = DeploymentConstants.getTokenManager();
65
66      uint256 weightedValueOfTokens = ownedAssetsPrices[0].price * (address(this).balance - msg.
            value) * tokenManager.debtCoverage(tokenManager.getAssetAddress(nativeTokenSymbol, true
            )) / (10 ** 26);
67
68      if (ownedAssetsPrices.length > 0) {
69
70          for (uint256 i = 0; i < ownedAssetsPrices.length; i++) {
71              IERC20Metadata token = IERC20Metadata(tokenManager.getAssetAddress(ownedAssetsPrices
                    [i].asset, true));
72              weightedValueOfTokens = weightedValueOfTokens + (ownedAssetsPrices[i].price * token.
                    balanceOf(address(this)) * tokenManager.debtCoverage(address(token)) *
                    fixVaultDecimals(address(token)) / (10 ** token.decimals() * 1e8));
73          }
74      }
75      return weightedValueOfTokens;
76  }
```

Listing 2.109: contracts/facets/arbitrum/SolvencyFacetProdArbitrum.sol

```
288 function _getTWVOwnedAssets(AssetPrice[] memory ownedAssetsPrices) internal virtual view
        returns (uint256) {
289     bytes32 nativeTokenSymbol = DeploymentConstants.getNativeTokenSymbol();
290     ITokenManager tokenManager = DeploymentConstants.getTokenManager();
291
292     uint256 weightedValueOfTokens = ownedAssetsPrices[0].price * address(this).balance *
            tokenManager.debtCoverage(tokenManager.getAssetAddress(nativeTokenSymbol, true)) / (10
            ** 26);
293
294     if (ownedAssetsPrices.length > 0) {
295
296         for (uint256 i = 0; i < ownedAssetsPrices.length; i++) {
297             IERC20Metadata token = IERC20Metadata(tokenManager.getAssetAddress(ownedAssetsPrices
                    [i].asset, true));
298             weightedValueOfTokens = weightedValueOfTokens + (ownedAssetsPrices[i].price * token.
                    balanceOf(address(this)) * tokenManager.debtCoverage(address(token)) / (10 **
                    token.decimals() * 1e8));
299         }
```

```
300         }
301         return weightedValueOfTokens;
302     }
```

Listing 2.110: contracts/facets/SolvencyFacetProd.sol

Impact This could lead to inconsistent asset valuations across different chains.

Suggestion Unify the handling of `msg.value` in the contracts `SolvencyFacetProdArbitrum` and `SolvencyFacetProdAvalanche`.

2.3.11 Remove redundant `view` functions in the `DiamondStorageLib` contract

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `DiamondStorageLib` contract, there are two contracts, i.e., `getTjV2OwnedBins` and `getTjV2OwnedBinsView`, that perform the same logic, only reading from storage. The only difference is the presence of the `view` modifier, which does not affect their functionality. Similarly, the functions `getUV3OwnedTokenIds` and `getUV3OwnedTokenIdsView` follow the same design. It is recommended to remove the redundant functions and retain only one of each with the appropriate `view` modifier.

```
175     function getTjV2OwnedBins() internal returns(ITraderJoeV2Facet.TraderJoeV2Bin[] storage bins){
176         TraderJoeV2Storage storage tjv2s = traderJoeV2Storage();
177         bins = tjv2s.ownedTjV2Bins;
178     }
179
180     function getTjV2OwnedBinsView() internal view returns(ITraderJoeV2Facet.TraderJoeV2Bin[]
            storage bins){
181         TraderJoeV2Storage storage tjv2s = traderJoeV2Storage();
182         bins = tjv2s.ownedTjV2Bins;
183     }
```

Listing 2.111: contracts/lib/DiamondStorageLib.sol

```
185     function getUV3OwnedTokenIds() internal returns(uint256[] storage tokenIds){
186         UniswapV3Storage storage uv3s = uniswapV3Storage();
187         tokenIds = uv3s.ownedUniswapV3TokenIds;
188     }
189
190     function getUV3OwnedTokenIdsView() internal view returns(uint256[] storage tokenIds){
191         UniswapV3Storage storage uv3s = uniswapV3Storage();
192         tokenIds = uv3s.ownedUniswapV3TokenIds;
193     }
```

Listing 2.112: contracts/lib/DiamondStorageLib.sol

Impact N/A

Suggestion Remove the redundant `view` functions.

2.3.12 Remove the redundant price query

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `GmxV2Facet` and `GmxV2PlusFacet` contracts, the function `_deposit` performs a redundant price query (i.e., `SolvencyMethods.getPrices(dataFeedIds)`). It is recommended to remove the redundant query for gas optimization.

```
68    bytes32[] memory dataFeedIds = new bytes32[](2);
69    dataFeedIds[0] = tokenManager.tokenAddressToSymbol(gmToken);
70    dataFeedIds[1] = tokenManager.tokenAddressToSymbol(depositedToken);
71
72    uint256 gmTokenUsdPrice = SolvencyMethods.getPrices(dataFeedIds)[0];
73    uint256 depositTokenUsdPrice = SolvencyMethods.getPrices(dataFeedIds)[1];
```

Listing 2.113: contracts/facets/GmxV2Facet.sol

```
70    bytes32[] memory dataFeedIds = new bytes32[](2);
71    dataFeedIds[0] = tokenManager.tokenAddressToSymbol(gmToken);
72    dataFeedIds[1] = tokenManager.tokenAddressToSymbol(depositedToken);
73
74    uint256 gmTokenUsdPrice = SolvencyMethods.getPrices(dataFeedIds)[0];
75    uint256 depositTokenUsdPrice = SolvencyMethods.getPrices(dataFeedIds)[1];
```

Listing 2.114: contracts/facets/GmxV2PlusFacet.sol

Impact The redundant price queries could result in unnecessary gas consumption.

Suggestion Remove the redundant price query.

2.3.13 Remove redundant validation checks in the `mintAndStakeGlp` function

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `mintAndStakeGlp` function of both the `GLPFacet` and `GLPFacetArbi` contracts, there is a redundant validation: `require(glpOutputAmount >=_minGlp, "Insufficient output amount")`. Specifically, this validation is already performed in the `GlpManager` contract during the invocation of `glpRewarder.mintAndStakeGlp`. It is recommended to remove this redundant validation for gas optimization.

```
49    function mintAndStakeGlp(address _token, uint256 _amount, uint256 _minUsdg, uint256 _minGlp)
          external nonReentrant onlyOwner noBorrowInTheSameBlock remainsSolvent{
50        // ...
51
52        uint256 glpOutputAmount = glpRewarder.mintAndStakeGlp(_token, _amount, _minUsdg, _minGlp);
53
54        require((glpToken.balanceOf(address(this)) - glpInitialBalance) == glpOutputAmount, "GLP
              minted and balance difference mismatch");
55        require(glpOutputAmount >=_minGlp, "Insufficient output amount");
```

Listing 2.115: contracts/facets/avalanche/GLPFacet.sol

```
50  function mintAndStakeGlp(address _token, uint256 _amount, uint256 _minUsdg, uint256 _minGlp)
        external nonReentrant onlyOwner noBorrowInTheSameBlock remainsSolvent{
51      // ...
52
53      uint256 glpOutputAmount = glpRewarder.mintAndStakeGlp(_token, _amount, _minUsdg, _minGlp);
54
55      require((glpToken.balanceOf(address(this)) - glpInitialBalance) == glpOutputAmount, "GLP
            minted and balance difference mismatch");
56      require(glpOutputAmount >=_minGlp, "Insufficient output amount");
```

Listing 2.116: contracts/facets/arbitrum/GLPFacetArbi.sol

Impact The redundant validation checks could result in unnecessary gas consumption.

Suggestion Remove the redundant validation checks.

2.3.14 Refactor the misleading annotation in the `getLiquidityTokenAmounts` function

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description The annotation of the `getLiquidityTokenAmounts` function in the `sPrimeImpl` contract states that the input `poolPrice` is an Oracle Price. However, based on the invocation trace, the input `poolPrice` is determined via the `getPoolPrice` function, which returns the pool's spot price.

```
45  /**
46  * @dev Returns the token balances for the specific bin.
47  * @param depositIds Deposited bin id list.
48  * @param liquidityMinted Liquidity minted for each bin.
49  * @param poolPrice Oracle Price
50  */
51  function getLiquidityTokenAmounts(uint256[] memory depositIds, uint256[] memory liquidityMinted
        , uint256 poolPrice) public view returns(uint256 amountX, uint256 amountY) {
```

Listing 2.117: contracts/token/sPrimeImpl.sol

```
248 function getPoolPrice() public view returns(uint256) {
249     uint256 price = PriceHelper.convert128x128PriceToDecimal(lbPair.getPriceFromId(lbPair.
            getActiveId()));
250     // price * 1e8 * 1edx / 1edy / 1e18
251     if (tokenXDecimals >= 10 + tokenYDecimals) {
252         price = price * 10 ** (tokenXDecimals - 10 - tokenYDecimals);
253     } else {
254         price = price / 10 ** (10 + tokenYDecimals - tokenXDecimals);
255     }
256     return price;
257 }
```

Listing 2.118: contracts/token/sPrime.sol

Impact N/A

Suggestion Refactor the annotation for better clarity.

2.3.15 Add access control checks to the `refundExecutionFee` function

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the `GmxV2CallbacksFacet` contract, the function `refundExecutionFee` is used for the gas refund callback from the GmxV2 protocol. However, there is no access control for this function, allowing anyone to invoke `refundExecutionFee` to wrap native tokens for any prime accounts. It is recommended to add the `nonReentrant` and `onlyGmxV2Keeper` modifiers to prevent potential security risks.

```
204   function refundExecutionFee(bytes32 /* key */, EventUtils.EventLogData memory /* eventData */)
          external payable {
205       wrapNativeToken();
206
207       emit GasFeeRefunded(msg.value);
208   }
```

Listing 2.119: contracts/facets/GmxV2CallbacksFacet.sol

Impact The lack of access control checks allows anyone to wrap native tokens for any prime accounts.

Suggestion Add the `nonReentrant` and `onlyGmxV2Keeper` modifiers to this function.

2.3.16 Allow a customizable `callbackGasLimit` in the `GmxV2Facet` and `GmxV2PlusFacet` contracts

Status Confirmed

Introduced by `Version 1-1`

Description In the `GmxV2Facet` and `GmxV2PlusFacet` contracts, the parameter `callbackGasLimit` is hardcoded to `500,000`. This hardcoded value could potentially cause callback execution to fail in the future. It is recommended to allow a customizable value for better flexibility.

```
44   data[2] = abi.encodeWithSelector(
45       IDepositUtils.createDeposit.selector,
46       IDepositUtils.CreateDepositParams({
47           receiver: address(this), //receiver
48           callbackContract: address(this), //callbackContract
49           uiFeeReceiver: address(0), //uiFeeReceiver
50           market: gmToken, //market
51           initialLongToken: marketToLongToken(gmToken), //initialLongToken
52           initialShortToken: marketToShortToken(gmToken), //initialShortToken
53           longTokenSwapPath: new address[](0), //longTokenSwapPath
54           shortTokenSwapPath: new address[](0), //shortTokenSwapPath
55           minMarketTokens: minGmAmount, //minMarketTokens
56           shouldUnwrapNativeToken: false, //shouldUnwrapNativeToken
57           executionFee: executionFee, //executionFee
```

```
58          callbackGasLimit: 500000 //callbackGasLimit
59      })
60  );
```

Listing 2.120: contracts/facets/GmxV2Facet.sol

```
46  data[3] = abi.encodeWithSelector(
47      IDepositUtils.createDeposit.selector,
48      IDepositUtils.CreateDepositParams({
49          receiver: address(this), //receiver
50          callbackContract: address(this), //callbackContract
51          uiFeeReceiver: address(0), //uiFeeReceiver
52          market: gmToken, //market
53          initialLongToken: depositedToken, //initialLongToken
54          initialShortToken: depositedToken, //initialShortToken
55          longTokenSwapPath: new address[](0), //longTokenSwapPath
56          shortTokenSwapPath: new address[](0), //shortTokenSwapPath
57          minMarketTokens: minGmAmount, //minMarketTokens
58          shouldUnwrapNativeToken: false, //shouldUnwrapNativeToken
59          executionFee: executionFee, //executionFee
60          callbackGasLimit: 500000 //callbackGasLimit
61      })
62  );
```

Listing 2.121: contracts/facets/GmxV2PlusFacet.sol

Impact The hardcoded value could potentially cause callback execution failure in the future.

Suggestion Allow a customizable `callbackGasLimit`.

2.3.17 Add whitelist checks while interacting with external contracts

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the protocol, several functions lack vital whitelist checks while interacting with third-party contracts, which are specified by the function inputs:

1. In the `TraderJoeV2Facet` contract, the function `removeLiquidityTraderJoeV2` lacks validation on whether the `lbPair` is whitelisted by the protocol. It is advised to implement such checks to avoid potential unauthorized interaction.

```
214     function removeLiquidityTraderJoeV2(ILBRouter traderJoeV2Router,
            RemoveLiquidityParameters memory parameters) external nonReentrant
            onlyOwnerOrInsolvent noBorrowInTheSameBlock {
215         if (!isRouterWhitelisted(address(traderJoeV2Router))) revert
                TraderJoeV2RouterNotWhitelisted();
216         ILBPair lbPair = ILBPair(traderJoeV2Router.getFactory().getLBPairInformation(
                parameters.tokenX, parameters.tokenY, parameters.binStep).LBPair);
217         lbPair.approveForAll(address(traderJoeV2Router), true);
```

Listing 2.122: contracts/facets/TraderJoeV2Facet.sol

2. In the PenpieFacet contract, the function claimRewards should apply a whitelist check on the market address.

```
249     function claimRewards(address market) external onlyOwner {
250         (
251             uint256 pendingPenpie,
252             address[] memory bonusTokenAddresses,
253             uint256[] memory pendingBonusRewards
254         ) = pendingRewards(market);
255         address[] memory stakingTokens = new address[](1);
256         stakingTokens[0] = market;
257         IMasterPenpie(MASTER_PENPIE).multiclaim(stakingTokens);
258
259         _handleRewards(pendingPenpie, bonusTokenAddresses, pendingBonusRewards);
260     }
```

Listing 2.123: contracts/facets/arbitrum/PenpieFacet.sol

Impact Allowing interaction with unverified pairs or markets may result in unexpected consequences.

Suggestion Add whitelist checks for the input addresses.

2.3.18 Revise the reward handling logic in the TraderJoeV2Facet contract

Status Fixed in Version 2

Introduced by Version 1-1

Description In the protocol's integration, many reward harvesting functions keep rewards in users' prime accounts and only transfer tokens to the prime account owners when the reward tokens are inactive (e.g., the function _handleRewards of the contract PenpieFacet). However, the contract TraderJoeV2Facet deviates from this pattern. Specifically, its claimReward function transfers rewards directly to prime account owners, regardless of the token status. It is recommended to standardize the reward handling logic across the third-party integration facets.

```
304   function _handleRewards(
305       uint256 pendingPenpie,
306       address[] memory bonusTokenAddresses,
307       uint256[] memory pendingBonusRewards
308   ) internal {
309       ITokenManager tokenManager = DeploymentConstants.getTokenManager();
310       address owner = DiamondStorageLib.contractOwner();
311
312       if (pendingPenpie > 0 && tokenManager.isTokenAssetActive(PNP)) {
313           _increaseExposure(tokenManager, PNP, pendingPenpie);
314       } else if (pendingPenpie > 0) {
315           PNP.safeTransfer(owner, pendingPenpie);
316       }
317
318       uint256 len = bonusTokenAddresses.length;
319       for (uint256 i; i != len; ++i) {
320           address bonusToken = bonusTokenAddresses[i];
321           uint256 pendingReward = pendingBonusRewards[i];
```

```
322             if (pendingReward == 0) {
323                 continue;
324             }
325
326             if (tokenManager.isTokenAssetActive(bonusToken)) {
327                 _increaseExposure(tokenManager, bonusToken, pendingReward);
328             } else {
329                 bonusToken.safeTransfer(owner, pendingReward);
330             }
331         }
332     }
```

Listing 2.124: contracts/facets/arbitrum/PenpieFacet.sol

```
71  function claimReward(IRewarder.MerkleEntry[] calldata merkleEntries) external nonReentrant
        onlyOwner {
72      uint256 length = merkleEntries.length;
73      IERC20[] memory tokens = new IERC20[](length);
74      uint256[] memory beforeBalances = new uint256[](length);
75      for (uint256 i; i != length; ++i) {
76          tokens[i] = merkleEntries[i].token;
77          beforeBalances[i] = tokens[i].balanceOf(address(this));
78      }
79
80      IRewarder rewarder = IRewarder(REWARDER);
81      rewarder.batchClaim(merkleEntries);
82
83      for (uint256 i; i != length; ++i) {
84          uint256 newBalance = tokens[i].balanceOf(address(this));
85          if (newBalance > beforeBalances[i]) {
86              address(tokens[i]).safeTransfer(msg.sender, newBalance - beforeBalances[i]);
87          }
88      }
89  }
```

Listing 2.125: contracts/facets/TraderJoeV2Facet.sol

Impact Users can directly claim rewards to their EOAs.

Suggestion Revise the claim reward logic accordingly.

2.3.19 Revise the contracts inheriting from the `ReentrancyGuardKeccak` contract

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `ReentrancyGuardKeccak`, the modifier `nonReentrant` is introduced to prevent reentrancy risks. However, the contracts `SmartLoanViewFacet` and `UniswapV2DEXFacet` inherit from `ReentrancyGuardKeccak` but do not utilize the corresponding modifier and storage slots.

It is recommended to revise these contracts to either remove the redundant inheritance or add the `nonReentrant` modifier for additional safety. For example, the developer could consider removing the inheritance in the contract `SmartLoanViewFacet` and adding the `nonReentrant`

modifier to the functions in the contract `UniswapV2DEXFacet`, which is further inherited by the contracts `SushiSwapDEXFacet`, `TraderJoeDEXFacet`, and `PangolinDEXFacet`.

```
15  contract SmartLoanViewFacet is ReentrancyGuardKeccak, SolvencyMethods {
```

Listing 2.126: contracts/facet/SmartLoanViewFacet.sol

```
17  contract UniswapV2DEXFacet is ReentrancyGuardKeccak, SolvencyMethods, OnlyOwnerOrInsolvent {
```

Listing 2.127: contracts/facet/UniswapV2DEXFacet.sol

Impact N/A

Suggestion Revise code logic accordingly.

2.3.20 Add validation checks for the `msg.value` and `executionFee`

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contracts `GmxV2Facet` and `GmxV2PlusFacet`, the `_deposit` and `_withdraw` functions organize three function calls to create deposit and withdrawal orders via `multicall`. In the first call, the execution fee (i.e., `msg.value`) is transferred to the GmxV2's exchange router [1] and is further wrapped based on the input parameter `executionFee`. However, there is a lack of validation between `msg.value` and `executionFee`, which could lead to unexpected consequences. As a result:

- If the `executionFee` is greater than `msg.value`, the order creation transaction will revert due to insufficient fees.
- If the `executionFee` is less than `msg.value`, the extra native token will be donated to the GmxV2's exchange router.

```
19  function _deposit(
20      address gmToken,
21      address depositedToken,
22      uint256 tokenAmount,
23      uint256 minGmAmount,
24      uint256 executionFee
25  ) internal nonReentrant noBorrowInTheSameBlock onlyOwner {
26      // ...
27      data[2] = abi.encodeWithSelector(
28          IDepositUtils.createDeposit.selector,
29          IDepositUtils.CreateDepositParams({
30              receiver: address(this), //receiver
31              callbackContract: address(this), //callbackContract
32              uiFeeReceiver: address(0), //uiFeeReceiver
33              market: gmToken, //market
34              initialLongToken: marketToLongToken(gmToken), //initialLongToken
35              initialShortToken: marketToShortToken(gmToken), //initialShortToken
36              longTokenSwapPath: new address[](0), //longTokenSwapPath
37              shortTokenSwapPath: new address[](0), //shortTokenSwapPath
```

[1] https://arbiscan.deth.net/address/0x900173a66dbd345006c51fa35fa3ab760fcd843b

```
38              minMarketTokens: minGmAmount, //minMarketTokens
39              shouldUnwrapNativeToken: false, //shouldUnwrapNativeToken
40              executionFee: executionFee, //executionFee
41              callbackGasLimit: 500000 //callbackGasLimit
42          })
43      );
44
45      depositedToken.safeApprove(getGmxV2Router(), 0);
46      depositedToken.safeApprove(getGmxV2Router(), tokenAmount);
47      BasicMulticall(getGmxV2ExchangeRouter()).multicall{value: msg.value}(data);
```

Listing 2.128: contracts/facet/GmxV2Facet.sol

Impact Mismatched `msg.value` and `executionFee` could cause transaction reversion or unintended donation of extra native tokens.

Suggestion Add validation checks for the `msg.value` and `executionFee`.

2.3.21 Implement the redeem logic in the `GogoPoolFacet` contract

Status Fixed in `Version 2`

Introduced by `Version 1-1`

Description In the contract `GogoPoolFacet`, the function `swapToGgAvax` enables users to deposit wrapped `AVAX` for `ggAVAX` tokens via the function `depositAVAX` of the contract `ggAvax`. However, the current implementation lacks a corresponding redeem functionality (e.g., the function `redeemAVAX` of the contract `ggAvax`).

Impact May prevent prime accounts from redeeming their `ggAVAX` tokens on GoGoPool.

Suggestion Implement the redeem logic in the `GogoPoolFacet` contract.

2.4 Note

2.4.1 Potential centralization risks

Introduced by `Version 1-1`

Description The protocol includes several privileged functions, such as `addPoolAssets` and `removeTokenAssets`. Additionally, all liquidator accounts are controlled by the team for performing liquidations. If the private key of the owner or a liquidator is lost or maliciously exploited, it could potentially cause losses to users.

2.4.2 Reentrancy risks in pools for tokens with callback mechanisms

Introduced by `Version 1-1`

Description The protocol should be aware of the potential risks when creating pools for tokens with a callback mechanism. In particular, pools involving tokens with a callback mechanism could introduce reentrancy vulnerabilities, potentially leading to losses.

2.4.3 Potential integration risks

Introduced by Version 1-1

Description The protocol integrates several third-party protocols, such as GmxV2, TraderJoeV2, and YakSwap. The team should remain vigilant about updates to these dependencies to mitigate potential security risks arising from inconsistent interaction patterns. It is important to note that updates to these external dependencies are not covered in this audit.

2.4.4 Potential deployment risks

Introduced by Version 1-1

Description In the protocol, several external dependencies (i.e., the smart contract addresses of third-party protocols) are hardcoded into the smart contracts for interaction. When deploying the protocol to different chains, the team must ensure the correct functionality and security by replacing the hardcoded addresses with the appropriate ones. Notably, this audit focuses solely on verifying the external dependencies deployed on the Arbitrum and Avalanche chains.

2.4.5 Malicious transfer of `sPrime` prevents users from receiving shares with their desired `tokenId`

Introduced by Version 1-1

Description In the contract `sPrime`, each user is allowed to have only one `tokenId` stored at index zero in the contract `PositionManager`. However, this design allows a malicious user to transfer a small amount of shares to a victim user, creating a position at index zero. As a result, the victim user cannot receive shares from others and must empty the existing position before receiving new shares with a desired `tokenId`.

```
229   function getUserTokenId(address user) public view returns(uint256 tokenId){
230       if(positionManager.balanceOf(user) > 0) {
231           tokenId = positionManager.tokenOfOwnerByIndex(user, 0);
232       }
233   }
```

Listing 2.129: contracts/token/sPrime.sol

```
674   function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual
          override {
675       if(from != address(0) && to != address(0)) {
676           uint256 lockedBalance = getLockedBalance(from);
677           uint256 fromBalance = balanceOf(from);
678           if (fromBalance < amount + lockedBalance) {
679               revert InsufficientBalance();
680           }
681           if (getUserTokenId(to) != 0) {
682               revert UserAlreadyHasPosition();
683           }
```

Listing 2.130: contracts/token/sPrime.sol

2.4.6 Potential withdrawal failure in pools with rebasing, inflation, or deflation tokens

Introduced by Version 1-2

Description In the contract `Pool`, the withdrawal lock mechanism allows users to create a withdrawal intent with a specified amount and withdraw their assets after 24 hours. In the `withdraw` function, there is a `require` check to ensure that the provided withdrawal amount (i.e., `_amount`) matches the proposed intent amount. However, when the pool contains a token with a rebasing, inflation, or deflation mechanism, the withdrawal intent amount may not match the actual amount the user can withdraw, which could result in a failed withdrawal.

```
499   function withdraw(uint256 _amount, uint256 intentIndex) external nonReentrant {
500       WithdrawalIntent[] storage intents = withdrawalIntents[msg.sender];
501       require(intentIndex < intents.length, "Invalid intent index");
502
503       WithdrawalIntent storage intent = intents[intentIndex];
504       require(intent.amount == _amount, "Withdrawal amount must match intent amount");
```

Listing 2.131: contracts/Pool.sol

2.4.7 Deprecated contracts

Introduced by Version 1-1

Description Certain contracts within the audit scope have been deprecated and are no longer in use. Below is a list of these deprecated contracts:

- deltaprime-primeloans/contracts/deployment/*
- deltaprime-primeloans/contracts/SmartLoansFactoryRestrictedAccess.sol
- deltaprime-primeloans/contracts/BtcEligibleUsersList.sol
- deltaprime-primeloans/contracts/facets/arbitrum/LTIPFacet.sol
- deltaprime-primeloans/contracts/facets/avalanche/CaiFacet.sol
- deltaprime-primeloans/contracts/facets/avalanche/SteakHutFinanceFacet.sol
- deltaprime-primeloans/contracts/facets/arbitrum/LevelFinanceFacet.sol
- deltaprime-primeloans/contracts/facets/avalanche/VectorFinanceFacet.sol
- deltaprime-primeloans/contracts/facets/AssetsExposureController.sol
- deltaprime-primeloans/contracts/facets/arbitrum/PenpieFacet.sol

2.4.8 Configure the variable `delay` properly

Introduced by Version 1-1

Description In the contract `Timelock`, the function `setDelay` is designed to update the variable `delay`, which must fall within the range defined by `MINIMUM_DELAY` and `MAXIMUM_DELAY`. However, this design carries a risk: setting `delay` to `MAXIMUM_DELAY` could effectively make the variable immutable, as it would no longer be possible to increase it further. The project must configure the variable `delay` appropriately to ensure the contract functions as intended.

Feedback from the project The project stated that the design is intentional.

2.4.9 Ensure proper synchronization of vPrime tokens

Introduced by Version 2

Description In the project, vPrime token balances represent users' voting power and are determined based on their positions (e.g., borrowed value). The `notifyVPrimeController` function is designed to synchronize users' vPrime tokens when their positions change. However, within the `AssetsOperationsFacet` contract, the `notifyVPrimeController` function is not triggered after debt swap operations, which impacts the accounting of users' voting power. Consequently, the project must promptly synchronize users' vPrime tokens to ensure accurate accounting of their voting power.

Feedback from the project The project stated that they are aware of this issue and decided to keep this design since the value change should be marginal (i.e., up to the slippage amount).

2.4.10 Ensure intentional withdrawal design in liquidation processes

Introduced by Version 2

Description In the contract `SmartLoanLiquidationFacet` , the funciton `liquidate` calculates the supply amount (i.e., the `supplyAmount` variable), which serves as the liquidator's repaid amount. The supply amount is determined as the difference between the total repaid amount and the token balance (i.e., `balanceOf(address(this))`).

However, this calculation of the supply amount does not account for users' withdrawal intents. Specifically, it does not use users' available balance in the computation. As a result, users' withdrawal intents remain in the `WithdrawalIntentFacet` contract and can be used at a later time. The project must implement proper checks to ensure intentional withdrawal design functions as intended during liquidation processes.

Feedback from the project The project stated that they decided to maintain this design and plan to design a liquidation mechanism by removing the use of the supply amount in the future version.

2.4.11 Integrate the RedStone oracle with a proper setting

Introduced by Version 1-1

Description The project integrates the RedStone oracle for price reference. Specifically, the solvency check function invokes the function `getOracleNumericValuesFromTxMsg` (inherited from RedStone oracle's consumer base contract) to determine a loan's status (e.g., whether it is liquidatable). However, due to the design of the `RedstoneDefaultsLib` contract, users can execute borrowing and repayment actions using "optimal" price data within a specific time window (e.g., `[block.timestamp - 3 minutes, block.timestamp + 1 minutes]`). Consequently, the project must integrate the RedStone oracle with appropriate time range settings.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS