Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Parallel

Veridise Inc.

June 7, 2022

► **Prepared For:**

Yubo Ruan | Parallel Foundation
parallel.fi

► **Prepared By:**

Ben Mariano
Jon Stephens
Kostas Ferles
Andreea Buterchi
Chaofan Shou

► **Contact Us:** contact@veridise.com

► **Version History:**

June 7, 2022	V1
May 5, 2022	Draft

Contents

1 Executive Summary

From May 3 to June 3, Parallel engaged Veridise to review the security of their OMNI Money Market Protocol. The review covered tokenization (focusing on the new NToken and MintableIncentivizedERC721), core protocol logic (such as borrowing/supplying mechanisms, liquidation logic, and pool/reserve behavior), as well as helpers and configuration files. Veridise conducted this assessment over 3 person-months, with three engineers working on code from commit 19d718c to e753591 of the `parallel-finance/omni-mm` repository. The auditing strategy involved tool-assisted analysis of the source code performed by Veridise engineers. The tools that were used in the audit included a combination of static analysis, bounded model checking, and formal verification. Some of these tools were developed specifically for the purpose of performing a thorough audit of the OMNI-MM contracts.

Summary of issues detected. The audit uncovered 34 issues, 14 of which are assessed to be of high or critical severity by Veridise auditors. Bugs discovered by Veridise can lead to a variety of undesired behaviors of the Parallel protocol, including dropping collateral (V-PAR-VUL-002), illegal withdrawing of collateral (V-PAR-VUL-004, V-PAR-VUL-006), illegal withdraw from an inactive pool (V-PAR-VUL-011), and liquidation denial of service (V-PAR-VUL-007, V-PAR-VUL-008, V-PAR-VUL-010). In addition to the high-severity bugs found, Veridise auditors also discovered a number of moderate severity issues, such as the possibility of charging users twice for liquidating an NFT (V-PAR-VUL-018) as well as a number of of code optimizations and suggestions for better code maintainability.

Code assessment. The OMNI Money Market Protocol (Omni-MM) is a fork of the AAVE V3 protocol and shares much of the same infrastructure from that project. Like AAVE, Omni-MM is a pool-based lending protocol that enables lenders to provide liquidity to pools and borrowers to borrow funds from the pools by using collateral. The key difference between Omni-MM and AAVE is that Omni-MM extends the protocol so that users can borrow against NFTs put up as collateral. This extension required new reasoning about the supplying, borrowing, and liquidation logic as well as the introduction of a new coin, the NToken, which users receive upon depositing NFTs into a reserve. Similar to ATokens in the AAVE protocol, NTokens are minted upon deposit of an NFT and they can be used as collateral until they are burned/redeemed/liquidated.

The Omni-MM implementation includes a test set which achieves decent coverage of the codebase as a whole. However, many of the tests are inherited from the AAVE protocol and thus only test the behavior of the AAVE-portion of the protocol (as opposed to the NFT logic added by Omni-mm). While the protocol is complicated, the Omni-MM additions follow the same style and conventions as the original AAVE code, making AAVE's documentation useful in understanding the desired behavior. Additionally, the Parallel engineering team shared additional preliminary documentation with us on their additions what are helpful for understanding the code.

Code Stability. Over the period of the audit, new code was pushed to the repository 99 times, with the most recent commit occurring on June 3. Many of these commits changed some behaviors of or added new features to the protocol. The Veridise auditors therefore had less time to review some of the modifications that were made to the protocol.

Recommendations. In accordance with the contract between Parallel and Veridise, our audit focused on the portions of the code that were updated/added by the Parallel engineers thus far. However, there are a number of decisions that are outside the scope of this audit that have serious implications on the security of the protocol (V-PAR-VUL-013 and V-PAR-VUL-014). We suggest the decisions be made with a careful understanding of the risks involved as laid out in this report.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Omni Money Market	19d718c - e753591	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
May 3 - June 3, 2022	Manual & Tools	3	3 person-months

Table 2.3: Vulnerability Summary.

Name	Number	Fixed
Critical-Severity Issues	7	7
High-Severity Issues	7	7
Moderate-Severity Issues	6	6
Low-Severity Issues	8	8
Informational-Severity Issues	6	6
Undetermined-Severity Issues	0	0
TOTAL	34	34

Table 2.4: Category Breakdown.

Name	Number
Logic Error	12
Validation Error	5
Protocol Issue	5
Gas Optimizations	4
Uninitialized Variable	3
Code Consistency	3
Unused Function	2

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the Omni-MM lending protocol, particularly as it applies to NFTs. In our audit, we sought to answer the following questions:

- Can a user always recover an underlying asset presuming they have the appropriate balance, their health factor is large enough, and the reserve is active?
- Can a user ever borrow an amount without the appropriate collateral?
- Can a user inappropriately withdraw a collateralized asset which is being borrowed against?
- If a user repays some amount that they have borrowed, is their debt appropriately reduced?
- Can a reserve ever be locked permanently when it still contains funds?
- Can a user ever illegally interact with an inactive pool (i.e., supply to or borrow from an inactive pool)?
- Are liquidators able to start a liquidation when a user's health factor drops below the set threshold?
- Can a user's collateral ever be inappropriately liquidated (i.e., their health factor is above the threshold)?
- Can a user who falls below the health factor threshold inappropriately prevent liquidation of their collateral?
- Upon an NFT liquidation, are both the liquidator and the user being liquidated appropriately compensated?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to evaluate how the code behaves given unexpected inputs. To do this, we leveraged the Foundry and hardhat testing frameworks. We also extended the existing Hardhat unit tests using the property-based testing capabilities of the Foundry framework. This approach enabled us to generalize individual unit tests so that more general properties that could be checked via random sampling by the Foundry tool.

Scope. To understand the scope of the audit, we first reviewed the AAVE documentation and focused our efforts on understanding the difference between the AAVE protocol and the extensions proposed in Omni-MM. In this phase, our main goal was to understand how the protocol is intended to behave in the presence of ERC721. As much of the code is copied directly from the AAVE protocol (which has been audited multiple times) we focused our efforts on the portions of the code added by the Parallel developers.

In terms of the scope of the audit, the key components we considered include the following:

- Supply/Withdraw Mechanisms
- Pool/Reserve Logic
- Liquidation Logic
- FlashClaim Logic
- Validation Logic
- User Configuration
- Reserve Configuration
- Tokenization (NToken, MinitableIncentivizedERC721)

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Moderate
Likely	Warning	Low	Moderate	High
Very Likely	Low	Moderate	High	Critical

In this case, we judge the likelihood of a vulnerability as follows:

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows:

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-PAR-VUL-001	No Type Validation	Critical	Fixed
V-PAR-VUL-002	Dropped Collateral	Critical	Fixed
V-PAR-VUL-003	Uninitialized Variable	Critical	Fixed
V-PAR-VUL-004	Unused Function	Critical	Fixed
V-PAR-VUL-005	Incorrect Collateral Balance	Critical	Fixed
V-PAR-VUL-006	Signature Mismatch	Critical	Fixed
V-PAR-VUL-007	DoS in liquidationERC721	Critical	Fixed
V-PAR-VUL-008	Liquidation Transfer DoS	High	Fixed
V-PAR-VUL-009	Collateral not Reset	High	Intended Behavior
V-PAR-VUL-010	Illegal Token Burn	High	Fixed
V-PAR-VUL-011	Illegal Withdrawal	High	Fixed
V-PAR-VUL-012	FlashClaim Price Manipulation	High	Fixed
V-PAR-VUL-013	NFT Volatility	High	Acknowledged
V-PAR-VUL-014	Governance Vulnerability	High	Acknowledged
V-PAR-VUL-015	Inefficient State Update	Moderate	Intended Behavior
V-PAR-VUL-016	Trusted NFTs	Moderate	Acknowledged
V-PAR-VUL-017	Illegal Borrow	Moderate	Fixed
V-PAR-VUL-018	Liquidation Double Fee	Moderate	Fixed
V-PAR-VUL-019	Address Provider DoS	Moderate	Fixed
V-PAR-VUL-020	ERC721 Supply after Cap	Moderate	Fixed
V-PAR-VUL-021	Uninitialized Variable	Low	Fixed
V-PAR-VUL-022	No Type Validation	Low	Fixed
V-PAR-VUL-023	Wrong Interface	Low	Fixed
V-PAR-VUL-024	Restrictive Require	Low	Fixed
V-PAR-VUL-025	No Type Validation	Low	Fixed
V-PAR-VUL-026	ERC721 Liquidation Bonus	Low	Intended Behavior
V-PAR-VUL-027	Underflow on Borrow	Low	Intended Behavior
V-PAR-VUL-028	Forcing Collateral Reserve	Low	Acknowledged

4.1 Detailed Description of Bugs

In this section, we describe each uncovered vulnerability in more detail.

4.1.1 V-PAR-VUL-001: No Type Validation

Severity	Critical	**Commit**	709297a
Type	Validation Error	**Status**	Fixed
Files	contracts/protocol/libraries/logic/LiquidationLogic.sol		
Functions	executeLiquidationCall		

Description Since the asset type is not checked in executeLiquidationCall and all of the functions invoked on the xToken are shared by both NToken and PToken, it is possible to use this function to liquidate an ERC721 rather than executeERC721LiquidationCall. Should this occur, it is possible for more collateral to liquidated than should be allowed.

Recommendations Add checks to executeLiquidationCall and executeERC721LiquidationCall that check the asset type.

4.1.2 V-PAR-VUL-002: Dropped Collateral

Severity	Critical	Commit	381f174
Type	Logical Error	Status	Fixed
Files	contracts/protocol/libraries/logic/LiquidationLogic.sol		
Functions	executeERC721LiquidationCall		

Description When an ERC721 is purchased during a liquidation, it's possible for the ERC721's price to be greater than the amount of debt covered. In this case, the difference between the purchase price and the debt covered is supplied back to the ERC721's original owner as PTokens if they have more debt than what was liquidated. Since these tokens are the remaining balance of the purchased NFT, they are collateral since they were transitively used as collateral. However, if the owner already has supplied this token and has marked the reserve of these tokens as not collateral then the supplied tokens will also not be collateral. Thus, it is possible for someone to regain possession of part of their collateral while still having outstanding debt.

```
1   function executeERC721LiquidationCall(
2       mapping(address => DataTypes.ReserveData) storage reservesData,
3       mapping(uint256 => address) storage reservesList,
4       mapping(address => DataTypes.UserConfigurationMap) storage usersConfig,
5       DataTypes.ExecuteERC721LiquidationCallParams memory params
6    ) external {
7       ...
8       if (vars.collateralDiscountedPrice > toPrtocolAmount) {
9           if (vars.userGlobalTotalDebt > vars.actualDebtToLiquidate) {
10              SupplyLogic.executeSupply(
11                  reservesData,
12                  reservesList,
13                  userConfig,
14                  DataTypes.ExecuteSupplyParams({
15                      asset: params.liquidationAsset,
16                      amount: vars.collateralDiscountedPrice -
17                          vars.actualDebtToLiquidate -
18                          vars.liquidationProtocolFeeAmount,
19                      onBehalfOf: params.user,
20                      referralCode: 0
21                  })
22              );
23          } else {
24              IERC20(params.liquidationAsset).safeTransferFrom(
25                  msg.sender,
26                  params.user,
27                  vars.collateralDiscountedPrice - toPrtocolAmount
28              );
29          }
30      }
31      ...
32  }
```

Snippet 4.1: Location where funds are supplied back to the user

```
    function executeSupply(
        mapping(address => DataTypes.ReserveData) storage reservesData,
        mapping(uint256 => address) storage reservesList,
        DataTypes.UserConfigurationMap storage userConfig,
        DataTypes.ExecuteSupplyParams memory params
    ) external {
        ...

        if (isFirstSupply) {
            userConfig.setUsingAsCollateral(reserve.id, true);
            emit ReserveUsedAsCollateralEnabled(
                params.asset,
                params.onBehalfOf
            );
        }

        ...
    }
```

Snippet 4.2: Location where a reserve is marked as collateral only if *isFirstSupply*

Recommendations Mark the newly supplied funds as collateral. Note, this can be done by always marking the reserve as collateral but this approach can mark more than just the new funds as collateral.

4.1.3 V-PAR-VUL-003: Active Reserve Dropped

Severity	Critical	**Commit**	19d718c
Type	Uninitialized Variable	**Status**	Fixed
Files	contracts/protocol/libraries/tokenization/NToken.sol .../libraries/tokenization/base/MintableIncentivizedERC721.sol contracts/protocol/libraries/logic/PoolLogic.sol		
Functions	mint, burn, _mintMultiple, _burnMultipe, executeDropReserve		

Description Total supply is never incremented on mints or decremented on burns. Therefore, a reserve containing tokens can be dropped by calling executeDropReserve. This could lead to a situation where a user could lose access to funds they have in the reserve.

```
1   function validateDropReserve(
2       mapping(uint256 => address) storage reservesList,
3       DataTypes.ReserveData storage reserve,
4       address asset
5    ) internal view {
6       ...
7
8       require(
9           IERC20(reserve.xTokenAddress).totalSupply() == 0,
10          Errors.ATOKEN_SUPPLY_NOT_ZERO
11      );
12   }
```

Snippet 4.3: Location where *totalSupply* is used to determine if a reserve can be dropped

Recommendations Update totalSupply when tokens are minted and burned.

4.1.4 V-PAR-VUL-004: Illegal Collateral Withdrawal

Severity	Critical	Commit	273253f
Type	Unused Function	Status	Fixed
Files	contracts/protocol/libraries/tokenization/NToken.sol		
Functions	_transfer		

Description NToken defines _transfer(address, address, uint256, bool) similar to PToken. Unlike PToken, this _transfer is never called because _transfer is only ever called with 3 arguments. As a result, the pool never finalizes the transfer which in turn means a user's the health factor is never validated. This allows a user with a bad health factor to transfer their NToken to another user with a good health factor who can then withdraw the token from the pool. Since the original user retains ownership of their borrowed assets, this can be exploited to drain the pool of funds.

```
    function _transfer(
        address from,
        address to,
        uint256 tokenId,
        bool validate
    ) internal {
        address underlyingAsset = _underlyingAsset;

        uint256 fromBalanceBefore = balanceOf(from);
        uint256 toBalanceBefore = balanceOf(to);

        bool isUsedAsCollateral = _isUsedAsCollateral[tokenId];
        _transferCollaterizable(from, to, tokenId, isUsedAsCollateral);

        if (validate) {
            POOL.finalizeTransfer(
                underlyingAsset,
                from,
                to,
                isUsedAsCollateral,
                tokenId,
                fromBalanceBefore,
                toBalanceBefore
            );
        }

        // emit BalanceTransfer(from, to, tokenId, index); TODO emit a transfer event
    }
```

Snippet 4.4: Location where unused function *_transfer* calls *_finalizeTransfer*

Recommendations Ensure the correct version of _transfer is being called in cases where a health-factor validation is necessary.

4.1.5 V-PAR-VUL-005: Incorrect Collateral Balance

Severity	Critical	Commit	a7af59d
Type	Logical Error	Status	Fixed
Files	contracts/protocol/tokenization/base/MintableIncentivizedERC721.sol		
Functions	_transferCollaterizable		

Description Transferring collateral using NToken's _transfer will decrease the address from's collateral balance by two instead of one. This is because even though MintableIncentivizedERC721._transfer adjusts the collateral balance, a flag passed to _transferCollaterizable will cause the collateral to be reduced again. This bug can impact many parts of the protocol since it is common to fetch an NToken's collateral balance. One concrete side-effect though is that this can prevent an NToken from being transferred between users.

```
    function _transferCollaterizable(
        address from,
        address to,
        uint256 tokenId,
        bool isUsedAsCollateral
    ) internal virtual {
        MintableIncentivizedERC721._transfer(from, to, tokenId);

        if (isUsedAsCollateral) {
            _userState[from].collaterizedBalance -= 1;
            _userState[to].collaterizedBalance += 1;
        }
    }
```

Snippet 4.5: Location where collateral is adjusted in *_transferCollaterizable*

```
    function _transfer(
        address from,
        address to,
        uint256 tokenId
    ) internal virtual {
        ...

        if (_isUsedAsCollateral[tokenId]) {
            delete _isUsedAsCollateral[tokenId];
            _userState[from].collaterizedBalance -= 1;
        }

        ...
    }
```

Snippet 4.6: Location where collateral is adjusted in *_transfer*

Recommendations Update the collateral balance only in one function.

4.1.6 V-PAR-VUL-006: Function Signature Mismatch

Severity	Critical	**Commit**	a7af59d
Type	Unused Function	**Status**	Fixed
Files	contracts/protocol/tokenization/NToken.sol		
Functions	Ntoken._transfer		

Description The function _transfer(address, address, uint128) in NToken is intended to override _transfer(address, address, uint256) in MintableIncentivizedERC721, however the signatures do not match and therefore the function is not overridden. Since this function is intended to call NToken's _transfer(address, address, uint256, bool), similar to V-PAR-VUL-004 this can cause funds to be drained by the pool since the owner's health factor will not be checked.

```
function _transfer(
        address from,
        address to,
        uint128 amount
    ) internal {
        _transfer(from, to, amount, true);
    }
```

Snippet 4.7: Location where *_transfer* is declared with an incorrect signature

Recommendations Change the signature of NToken._transfer to match MintableIncentivizedERC721._transfer and add the override keyword to the function's header. Note, the third argument should also be named tokenId rather than amount since this is an ERC721.

4.1.7 V-PAR-VUL-007: DoS in liquidationERC721

Severity	Critical	Commit	e747ba6
Type	Logical Error	Status	Fixed
Files	contracts/protocol/libraries/logic/LiquidationLogic.sol		
Functions	_calculateERC721LiquidationParameters		

Description If an ERC721 is liquidated using an asset that the liquidated user is not currently borrowing from, then the transaction will revert. This is because vars.actualDebtToLiquidate is zero in this situation, causing vars.debtToCoverInBaseCurrency and vars.actualLiquidationBonus to also be zero. Since vars.actualLiquidationBonus eventually used as the denominator in a division, the transaction will revert due to a division by zero error.

```
    function _calculateERC721LiquidationParameters(
        DataTypes.ReserveData storage collateralReserve,
        DataTypes.ReserveCache memory debtReserveCache,
        address collateralAsset,
        address debtAsset,
        uint256 userGlobalTotalDebt,
        uint256 debtToCover,
        uint256 userCollateralBalance,
        uint256 liquidationBonus,
        IPriceOracleGetter oracle
    )
        internal view returns (uint256, uint256, uint256, uint256, uint256)
    {
        ...
        vars.actualLiquidationBonus = _calculateLiquidationBonus(
            liquidationBonus,
            vars.debtToCoverInBaseCurrency,
            userGlobalTotalDebt
        );

        vars.collateralDiscountedPrice = vars
            .collateralPriceInDebtAsset
            .percentDiv(vars.actualLiquidationBonus);
        ...
    }
```

Snippet 4.8: Location where the DoS occurrs due to *vars.actualLiquidationBonus* being 0

Recommendations Either set vars.collateralDiscountedPrice to vars.collateralPriceInDebtAsset in this case or adjust _calculateLiquidationBonus to return a minimum bonus when the covered debt is zero.

4.1.8 V-PAR-VUL-008: Liquidation Transfer DoS

Severity	High	**Commit**	a7af59d
Type	Logical Error	**Status**	Fixed
Files	contracts/protocol/tokenization/NToken.sol		
Functions	Ntoken._transferOnLiquidation		

Description Upon a liquidation, a user can choose to receive the xToken of the purchased asset. However, (assuming V-PAR-VUL-007 and V-PAR-VUL-006 are fixed) this process will always revert for an ERC721 because NToken's transfer will check the user's health factor. Since the user is being liquidated, it will be too low and therefore the pool will revert the transaction.

```
    function transferOnLiquidation(
        address from,
        address to,
        uint256 val
    ) external override onlyPool {
        // Being a normal transfer, the Transfer() and BalanceTransfer() are emitted
        // so no need to emit a specific event here
        _transfer(from, to, val);
    }
```

Snippet 4.9: Location where *transferOnLiquidation* calls the *_transfer* that validates a user's HF

Recommendations Change the body of transferOnLiquidation to _transfer(from, to, value, false).

4.1.9 V-PAR-VUL-009: Collateral Not Reset

Severity	High	Commit	273253f
Type	Logical Error	Status	Intended Behavior
Files	contracts/protocol/libraries/logic/SupplyLogic.sol		
Functions	executeUseReserveAsCollateral		

Description If a user marks a reserve as not being collateral with this function, it will only mark the reserve as not collateral but will not update the reserve's NTokens. The NToken can therefore have a collateral balance even though the reserve is not collateral. The user therefore has to use executeUseERC721AsCollateral to make adjustments to to each NToken to bring the collateral balance into line with the reserve's status.

```
    function executeUseReserveAsCollateral(
        mapping(address => DataTypes.ReserveData) storage reservesData,
        mapping(uint256 => address) storage reservesList,
        DataTypes.UserConfigurationMap storage userConfig,
        address asset,
        bool useAsCollateral,
        uint256 reservesCount,
        address priceOracle
    ) external {
        ...

        if (useAsCollateral) {
            userConfig.setUsingAsCollateral(reserve.id, true);
            emit ReserveUsedAsCollateralEnabled(asset, msg.sender);
        } else {
            userConfig.setUsingAsCollateral(reserve.id, false);
            ValidationLogic.validateHFAndLtv(
                reservesData,
                reservesList,
                userConfig,
                asset,
                msg.sender,
                reservesCount,
                priceOracle
            );

            emit ReserveUsedAsCollateralDisabled(asset, msg.sender);
        }
    }
```

Snippet 4.10: Location where a reserve is marked as collateral

Recommendations Modify each NToken for the user when the reserve's status is changed to adjust the collateral balance.

Developer Response This is intended behavior for this function. It is intended to be a gas-efficient way of marking all ERC721s in a reserve as not being collateral. An ERC721 is only considered collateral if the reserve is marked as collateral and the NToken is as well. This function can therefore be used to mark all NTokens in a reserve as not collateral without iterating over them. In addition, the user should enable the NTokens as collateral explicitly using executeUseERC721AsCollateral.

4.1.10 V-PAR-VUL-010: Illegal Token Burn

Severity	High	**Commit**	709297a
Type	Uninitailized Variable	**Status**	Fixed
Files	contracts/protocol/libraries/logic/LiquidationLogic.sol		
Functions	_burnCollateralNTokens		

Description The function _burnCollateralNTokens will burn the NToken with the tokenID vars.actualCollateralToLiquidate. However, this variable is not set in the call executeERC721LiquidationCall, and will therefore always be equal to 0. Thus, the only token that can ever be burnt in this manner is a token with token id 0. In most cases, this will revert if the user being liquidated does not own the token with id 0.

```
1    function _burnCollateralNTokens(
2        DataTypes.ReserveData storage collateralReserve,
3        DataTypes.ExecuteERC721LiquidationCallParams memory params,
4        LiquidationCallLocalVars memory vars
5    ) internal {
6        // Burn the equivalent amount of xToken, sending the underlying to the
     liquidator
7        uint256[] memory tokenIds = new uint256[](1);
8        tokenIds[0] = vars.actualCollateralToLiquidate;
9        INToken(vars.collateralXToken).burn(
10           params.user,
11           msg.sender,
12           tokenIds,
13           0
14       );
15   }
```

Snippet 4.11: Location where burn is called with the uninitialized variable *vars.actualCollateralToLiquidate*

Recommendations Instead of using vars.actualCollateralToLiquidate, use params.collateralTokenId.

4.1.11 V-PAR-VUL-011: Illegal Withdrawal

Severity	High	Commit	19d718c
Type	Logical Error	Status	Fixed
Files	contracts/protocol/libraries/logic/SupplyLogic.sol		
Functions	executeWithdrawERC721		

Description The function executeWithdrawERC721 does not check the reserve's status in the same way executeWithdraw does. Therefore, if a pool with ERC721 tokens is paused, a user will still be able to withdraw their funds which is inconsistent with the PToken behavior.

```
    function executeWithdrawERC721(
        mapping(address => DataTypes.ReserveData) storage reservesData,
        mapping(uint256 => address) storage reservesList,
        DataTypes.UserConfigurationMap storage userConfig,
        DataTypes.ExecuteWithdrawERC721Params memory params
    ) external returns (uint256) {
        DataTypes.ReserveData storage reserve = reservesData[params.asset];
        DataTypes.ReserveCache memory reserveCache = reserve.cache();

        reserve.updateState(reserveCache);
        uint256 amountToWithdraw = params.tokenIds.length;

        bool withdrwingAllCollateral = INToken(reserveCache.xTokenAddress).burn(
            msg.sender,
            params.to,
            params.tokenIds,
            reserveCache.nextLiquidityIndex
        );

        if (userConfig.isUsingAsCollateral(reserve.id)) {
            if (userConfig.isBorrowingAny()) {
                ValidationLogic.validateHFAndLtv(
                    reservesData, reservesList, userConfig, params.asset, msg.sender,
                    params.reservesCount, params.oracle
                );
            }

            if (withdrwingAllCollateral) {
                userConfig.setUsingAsCollateral(reserve.id, false);
                emit ReserveUsedAsCollateralDisabled(params.asset, msg.sender);
            }
        }

        emit WithdrawERC721(...);

        return amountToWithdraw;
    }
```

Snippet 4.12: Location of the executeWithdrawERC721 function

Recommendations Add a check into executeWithdrawERC721 that the reserve is active.

4.1.12 V-PAR-VUL-012: Flashclaim Price Manipulation

Severity	High	**Commit**	273253f
Type	Protocol Issue	**Status**	Fixed
Files	contracts/protocol/tokenization/NToken.sol		
Functions	flashClaim		

Description This function allows the owner of an ERC721 to temporarily regain ownership of their tokens for the space of a transaction. However, due to the diversity and volatility of ERC721 tokens, the user could perform an action that may affect the price of the asset and therefore should lead to a liquidation. However, the interface provided to perform a flashclaim is not unified with the rest of the protocol. The AAVE protocol requires the user to interact with the pool itself, rather than an individual xToken, partially to make it easy to find possible addresses that can be liquidated. In this case, it might escape the notice of liquidators that an address is insolvent and should be liquidated.

Recommendations Similar to withdraw/supply provide the functionality at the pool level and emit the FlashClaim event from the pool so that liquidators can easily find and check the health of an address.

4.1.13 V-PAR-VUL-013: NFT Price Volatility

Severity	High	**Commit**	NA
Type	Protocol Issue	**Status**	Acknowledged
Files		NA	
Functions		NA	

Description The protocol allows users to borrow against ERC721 tokens. These tokens commonly have fluid, difficult to predict and easily manipulated prices that make many of them high risk collateral assets. This could be used by a borrower to obtain a favorable loan with an asset that soon becomes worthless, providing no incentive for the borrower to pay back the loan. In addition, if the collateral asset's value is now greatly diminished it is likely that the funds could not be recovered through liquidation and therefore it may be impossible for the pool to return funds to all those who deposited the borrowed asset.

Alleviation The developers acknowledged this issue and stated that they were planning on mitigating the risks by:

- Only listing BlueChip NFTs (at least initially)
- Preferring oracles that use multiple sources of information such as those provided by Chainlink

4.1.14 V-PAR-VUL-014: Governance

Severity	High	**Commit**	NA
Type	Protocol Issue	**Status**	Acknowledged
Files		NA	
Functions		NA	

Description While the governance structure is out-of-scope for the current audit, we do notice that the structure of the governance could significantly impact the safety of the protocol. For instance, if a voting is proportional to a user's supply, the protocol could be vulnerable to a flashloan attack. If a malicious user were allowed to take control of the protocol, they could perform a number of harmful actions such as adding malicious tokens.

Recommendations Carefully construct the governance structure such that it is resilient to a small number of bad actors with a large amount of funds.

Alleviation The developers acknowledged this issue and stated that they would consider strategies that are robust against these types of attacks such as by implementing a quadratic voting based governance.

4.1.15 V-PAR-VUL-015: Inefficient State Update

Severity	Moderate	**Commit**	273253f
Type	Logical Error	**Status**	Intended Behavior
Files	contracts/protocol/libraries/logic/SupplyLogic.sol		
Functions	executeUseReserveAsCollateral		

Description This function requires that a user's collateralBalance be positive to set the reserve as collateral. It therefore cannot be used to set all of the NTokens in a reserve as collateral, instead they have to set them one at a time using executeUseERC721AsCollateral.

```
    function executeUseReserveAsCollateral(
        mapping(address => DataTypes.ReserveData) storage reservesData,
        mapping(uint256 => address) storage reservesList,
        DataTypes.UserConfigurationMap storage userConfig,
        address asset,
        bool useAsCollateral,
        uint256 reservesCount,
        address priceOracle
    ) external {
        ...

        if (reserveCache.assetType == DataTypes.AssetType.ERC20) {
            userBalance = IERC20(reserveCache.xTokenAddress).balanceOf(
                msg.sender
            );
        } else {
            userBalance = ICollaterizableERC721(reserveCache.xTokenAddress)
                .collaterizedBalanceOf(msg.sender);
        }

        ValidationLogic.validateSetUseReserveAsCollateral(
            reserveCache,
            userBalance
        );

        ...
    }
```

Snippet 4.13: Location where *collateralizedBalanceOf* is used

Recommendations If parameter useAsCollateral is set to true, then use the non-colletarized balance (total - colletarized).

Developer Response This is intended behavior for this function. It is intended to be a gas-efficient way of marking all ERC721s in a reserve as not being collateral. An ERC721 is only considered collateral if the reserve is marked as collateral and the NToken is as well. This function can therefore be used to mark all NTokens in a reserve as not collateral without iterating over them. In addition, the user should enable the NTokens as collateral explicitly using executeUseERC721AsCollateral.

4.1.16 V-PAR-VUL-016: Trusted NFTs

Severity	Moderate	**Commit**	bfc2d4b
Type	Protocol Issue	**Status**	Acknowledged
Files	contracts/protocol/libraries/logic/FlashClaimLogic.sol contracts/protocol/libraries/logic/SupplyLogic.sol		
Functions	executeFlashClaim, executeSupplyERC721		

Description If the protocol allows a malicious ERC721 token to be used as collateral, it is possible an attacker can borrow assets for free by building a backdoor into the token. Additionally, if a token is not intended to be malicious but has a bug, it can be exploited to a similar end. For example, it is currently assumed that a transfer will reset a token's approval (as stated by the ERC721 spec). If, however, this single line of code is missing from the token's transfer, this could be used to steal collateral tokens back.

Recommendations As part of the token approval process, ensure that the token's code is evaluated and is safe.

Alleviation The developers acknowledged this issue and stated that they were planning on mitigating the risks by listing mature BlueChip NFTs that have been shown to be robust against attacks.

4.1.17 V-PAR-VUL-017: Illegal ERC721 Borrow

Severity	Moderate	**Commit**	381f174
Type	Validation Error	**Status**	Fixed
Files	contracts/protocol/libraries/logic/BorrowLogic.sol		
Functions	executeBorrow		

Description If someone accidentally enables borrowing from an ERC721's reserve, executeBorrow will allow someone to borrow ERC721 tokens. Since ERC721s have a different model than ERC20s, if this happens it could allow someone to borrow another individual's collateral. They therefore couldn't be liquidated until the loan was repaid.

Recommendations Add in an asset type check which disallows this function to be called with ERC721 tokens.

4.1.18 V-PAR-VUL-018: Liquidation Double Fee

Severity	Moderate	Commit	e510840
Type	Logic Error	Status	Fixed
Files	contracts/protocol/libraries/logic/LiquidationLogic.sol		
Functions	executeERC721LiquidationCall		

Description Since a call to executeERC721LiquidationCall can succeed with a reserve id that params.user is not borrowing from, a borrow position with an NFT as collateral is subject to twice the amount of protocol fees and is essentially subjected to a price reduction twice (one as NTokens and one as PToken).

```
    function _calculateERC721LiquidationParameters(
        DataTypes.ReserveData storage collateralReserve,
        DataTypes.ReserveCache memory debtReserveCache,
        address collateralAsset,
        address debtAsset,
        uint256 userGlobalTotalDebt,
        uint256 debtToCover,
        uint256 userCollateralBalance,
        uint256 liquidationBonus,
        IPriceOracleGetter oracle
    ) internal view returns (uint256, uint256, uint256) {
        ...

        vars.liquidationProtocolFeePercentage = collateralReserve
            .configuration
            .getLiquidationProtocolFee();

        vars.collateralPriceInDebtAsset = ((vars.collateralPrice *
            vars.debtAssetUnit) /
            (vars.debtAssetPrice * vars.collateralAssetUnit));

        uint256 globalDebtPrice = (userGlobalTotalDebt / vars.debtAssetPrice) *
            BASE_CURRENCY_DECIMALS;

        vars.collateralDiscountedPrice = vars
            .collateralPriceInDebtAsset
            .percentDiv(liquidationBonus);

        ...
    }
```

Snippet 4.14: Location where the liquidation fee and collateral discount are computed

Recommendations Eliminate (or reduce) the protocol fee when a liquidator is not paying any debt.

4.1.19 V-PAR-VUL-019: Address Provider DoS

Severity	Moderate	**Commit**	6f1a895
Type	Logical Error	**Status**	Fixed
Files	contracts/protocol/configuration/PoolAddressesProviderRegistry.sol		
Functions	registerAddressesProvider		

Description Currently, when a poolAddressesProvider is registered, it is possible to set it to address (0x0). As long as its id is different from 0, it is added to the list of providers in the registry. However, no operation can be performed (e.g. setting the pool, the data provider, etc.) by the Provider, because deploying a poolAddressesProvider contract with its owner set to address (0x0) is not allowed (see transferOwnership from Ownable.sol).

```
1   function registerAddressesProvider(address provider, uint256 id)
2       external
3       override
4       onlyOwner
5   {
6       require(id != 0, Errors.INVALID_ADDRESSES_PROVIDER_ID);
7       require(
8           _idToAddressesProvider[id] == address(0),
9           Errors.INVALID_ADDRESSES_PROVIDER_ID
10      );
11      require(
12          _addressesProviderToId[provider] == 0,
13          Errors.ADDRESSES_PROVIDER_ALREADY_ADDED
14      );
15
16      ...
17  }
```

Snippet 4.15: Location where *registerAddressesProvider* omits a check on *provider*

Recommendations Check if the passed value for provider is not address (0x0).

4.1.20 V-PAR-VUL-020: ERC721 Supply after Cap

Severity	Moderate	**Commit**	e753591
Type	Logical Error	**Status**	Fixed
Files	/contracts/protocol/libraries/logic/ValidationLogic.sol		
Functions	validateSupply		

Description The protocol uses the supply cap to restrict the number of tokens that can be supplied to the pool. Initially this cap is set to zero, which equates to no cap. If the admin sets a supply cap for an ERC721 though, tokens can be provided even after it is exceeded. This can cause the pool to be flooded with risky collateral, which increases risk.

```
    function validateSupply(
        DataTypes.ReserveCache memory reserveCache,
        uint256 amount,
        DataTypes.AssetType assetType
    ) internal view {
        ...

        if (assetType == DataTypes.AssetType.ERC20) {
            uint256 supplyCap = reserveCache
                .reserveConfiguration
                .getSupplyCap();
            require(
                supplyCap == 0 ||
                    (IPToken(reserveCache.xTokenAddress)
                        .scaledTotalSupply()
                        .rayMul(reserveCache.nextLiquidityIndex) + amount) <=
                    supplyCap *
                        (10**reserveCache.reserveConfiguration.getDecimals()),
                Errors.SUPPLY_CAP_EXCEEDED
            );
        }
    }
```

Snippet 4.16: Location where *supplyCap* is checked for an ERC20, but omitted for an ERC721

Recommendations Even if it is desired not to have a cap for ERC721s, we would still recommend checking the supply cap since the default is an unlimited cap. This allows admins to change their mind if they grow concerned about collateral risk.

4.1.21 V-PAR-VUL-021: Incorrect Liquidation Logic Event

Severity	Low	Commit	709297a
Type	Uninitialized Variable	Status	Fixed
Files	contracts/protocol/libraries/logic/LiquidationLogic.sol		
Functions	executeERC721LiquidationCall		

Description The ERC721LiquidationCall event emitted at the end of executeERC721LiquidationCall includes vars.actualCollateralToLiquidate which is never set and therefore will always be 0.

```
    function executeERC721LiquidationCall(
        mapping(address => DataTypes.ReserveData) storage reservesData,
        mapping(uint256 => address) storage reservesList,
        mapping(address => DataTypes.UserConfigurationMap) storage usersConfig,
        DataTypes.ExecuteERC721LiquidationCallParams memory params
    ) external {
        ...
        emit ERC721LiquidationCall(
            params.collateralAsset,
            params.liquidationAsset,
            params.user,
            vars.actualDebtToLiquidate,
            vars.actualCollateralToLiquidate,
            msg.sender,
            params.receiveNToken
        );
    }
```

Snippet 4.17: Location where *ERC721LiquidationCall* is emitted with an uninitialized variable

Recommendations We suspect this should be params.collateralTokenID instead.

4.1.22 V-PAR-VUL-022: No Type Validation

Severity	Low	**Commit**	709297a
Type	Validation Error	**Status**	Fixed
Files	contracts/protocol/libraries/logic/LiquidationLogic.sol contracts/protocol/libraries/logic/SupplyLogic.sol		
Functions	executeERC721LiquidationCall, executeSupply, exeucteSupplyERC721, executeWithdraw, executeWithdrawERC721		

Description Currently if someone calls these functions with the wrong asset type, the call will revert once a function is invoked that is not shared by NToken and PToken. While this is the intended behavior, the user might not know why the revert occurred.

```
    function validateWithdraw(
        DataTypes.ReserveCache memory reserveCache,
        uint256 amount,
        uint256 userBalance
    ) internal pure {
        require(amount != 0, Errors.INVALID_AMOUNT);
        require(
            amount <= userBalance,
            Errors.NOT_ENOUGH_AVAILABLE_USER_BALANCE
        );

        (bool isActive, , , , bool isPaused) = reserveCache
            .reserveConfiguration
            .getFlags();
        require(isActive, Errors.RESERVE_INACTIVE);
        require(!isPaused, Errors.RESERVE_PAUSED);
    }
```

Snippet 4.18: Location where an ERC20 withdraw is validated without checking the asset type

Recommendations Add an asset type check and provide an informational error message if the wrong type of asset is provided. Note, this check would also simplify the validation logic since this asset type information could be assumed.

4.1.23 V-PAR-VUL-023: Wrong Interface

Severity	Low	**Commit**	709297a
Type	Code Consistency	**Status**	Fixed
Files	contracts/protocol/libraries/logic/LiquidationLogic.sol		
Functions	_liquidatePTokens		

Description _liquidatePTokens is intended to liquidate a PToken, but makes the call through the INToken interface. While this will result in the intended behavior since INToken and IPToken share the invoked function, this could lead to mistakes later on if INToken's interface is changed.

```
    function _liquidatePTokens(
        mapping(address => DataTypes.ReserveData) storage reservesData,
        mapping(uint256 => address) storage reservesList,
        mapping(address => DataTypes.UserConfigurationMap) storage usersConfig,
        DataTypes.ReserveData storage collateralReserve,
        DataTypes.ExecuteLiquidationCallParams memory params,
        LiquidationCallLocalVars memory vars
    ) internal {
        uint256 liquidatorPreviousPTokenBalance = IERC20(vars.collateralXToken)
            .balanceOf(msg.sender);
        INToken(vars.collateralXToken).transferOnLiquidation(
            params.user,
            msg.sender,
            vars.actualCollateralToLiquidate
        );

        ...
    }
```

Snippet 4.19: Location where the *INToken* interface is used rather than the *IPToken* interface

Recommendations Change the code to cast to IPToken and always cast to interfaces that the intended contract inherits from.

4.1.24 V-PAR-VUL-024: Restrictive Flashclaim Require

Severity	Low	**Commit**	381f174
Type	Logic Error	**Status**	Fixed
Files	contracts/protocol/libraries/logic/ValidationLogic.sol		
Functions	validateFlashClaim		

Description The validateFlashClaim function checks that !caller.isContract() which means that if anyone wanted to build contracts on top of the Omni Money Market (and therefore would be considered the owner of the token), they would not be able to use the flashClaim functionality.

```
    function validateFlashClaim(
        DataTypes.ReserveData storage reserve,
        DataTypes.ExecuteFlashClaimParams memory params
    ) internal view {
        address caller = msg.sender;
        require(!caller.isContract(), Errors.NOT_EOA);
        require(
            params.receiverAddress != address(0),
            Errors.ZERO_ADDRESS_NOT_VALID
        );

        // only token owner can do flash claim
        for (uint256 i = 0; i < params.nftTokenIds.length; i++) {
            require(
                INToken(reserve.xTokenAddress).ownerOf(params.nftTokenIds[i]) ==
                    caller,
                Errors.NOT_THE_OWNER
            );
        }
    }
```

Snippet 4.20: Location where flashClaim checks if the caller is a contract

Recommendations Remove this check unless the desired behavior is that certain contracts would be unable to use the flashClaim functionality.

4.1.25 V-PAR-VUL-025: Potential for flashClaim of ERC20

Severity	Low	**Commit**	381f174
Type	Validation Error	**Status**	Fixed
Files	contracts/protocol/libraries/logic/ValidationLogic.sol		
Functions	validateFlashClaim		

Description The validateFlashClaim function should check the assetType of the reserve to ensure someone is attempting to flashClaim an ERC721. Currently if someone attempted to flashClaim an ERC20, the call to ownerOf will revert, however if ownerOf is added to PToken in the future, flashClaim could be used to perform a flashLoan of ERC20 tokens.

```
    function validateFlashClaim(
        DataTypes.ReserveData storage reserve,
        DataTypes.ExecuteFlashClaimParams memory params
    ) internal view {
        address caller = msg.sender;
        require(!caller.isContract(), Errors.NOT_EOA);
        require(
            params.receiverAddress != address(0),
            Errors.ZERO_ADDRESS_NOT_VALID
        );

        // only token owner can do flash claim
        for (uint256 i = 0; i < params.nftTokenIds.length; i++) {
            require(
                INToken(reserve.xTokenAddress).ownerOf(params.nftTokenIds[i]) ==
                    caller,
                Errors.NOT_THE_OWNER
            );
        }
    }
```

Snippet 4.21: Location where a flashClaim is validated without checking the asset type

Recommendations Add an asset type check to flashClaim to ensure only ERC721 tokens can be flashClaimed.

4.1.26 V-PAR-VUL-026: ERC721 Liquidation Bonus

Severity	Low	**Commit**	e747ba6
Type	Logical Error	**Status**	Intended Behavior
Files	contracts/protocol/libraries/logic/LiquidationLogic.sol		
Functions	_calculateLiquidationBonus		

Description Liquidators are incentivized to liquidate collateral as a way of returning borrowed funds back to the pool. As it stands, however, ERC721 liquidators may not receive the full liquidation bonus even if they liquidate a full ERC721 token. The liquidation bonus is currently scaled by how much of a user's debt is being liquidated. Thus, if someone has a large debt with respect to the value of the ERC721 token, liquidators might receive little to no bonus at all.

```
    function _calculateLiquidationBonus(
        uint256 protocolMaxBonus,
        uint256 debtToBeRepaid,
        uint256 totalGlobalDebt
    ) internal view returns (uint256) {
        return (debtToBeRepaid * protocolMaxBonus) / totalGlobalDebt;
    }
```

Snippet 4.22: Location where the liquidation bonus is calculated

Recommendations If the intention is to provide a bonus based on how much of the ERC721 is liquidated, the denominator of the division should be the price of the ERC721 instead of totalGlobalDebt.

Developer Response The developers indicated that this was the intended behaviour because they want to incentivize a liquidator to liquidate the ERC721 that would pay back the largest amount of debt.

4.1.27 V-PAR-VUL-027: Underflow on Borrow

Severity	Low	Commit	e753591
Type	Validation Error	Status	Intended Behavior
Files	contracts/protocol/pool/DefaultReserveInterestRateStrategy.sol		
Functions	calculateInterestRates		

Description If a user attempts to borrow more tokens than is available in the pool, the transaction can revert from an arithmetic underflow in the function calculateInterestRates. While this is the intended behavior, it does not inform the user of why the revert occurred.

```
    function calculateInterestRates(
        DataTypes.CalculateInterestRatesParams calldata params
    ) external view override returns (uint256, uint256, uint256)
    {
        ...

        if (vars.totalDebt != 0) {
            vars.stableToTotalDebtRatio = params.totalStableDebt.rayDiv(
                vars.totalDebt
            );
            vars.availableLiquidity =
                IERC20(params.reserve).balanceOf(params.xToken) +
                params.liquidityAdded -
                params.liquidityTaken;

            ...
        }

        ...
    }
```

Snippet 4.23: Location where an underflow occurs if more funds are borrowed than available

Recommendations When validating a borrow, check that the pool has enough funds to fulfill the request. If it does not, revert with an informative message.

Developer Response Since this case cannot lead to harmful behaviors and is not likely to occur often the developers are not intending to fix this issue in the contract. They will include additional checks in the frontend to provide the necessary error reporting.

4.1.28 V-PAR-VUL-028: Forcing Collateral Reserve

Severity	Low	Commit	e753591
Type	Protocol Issue	Status	Acknowledged
Files	contracts/protocol/libraries/logic/LiquidationLogic.sol		
Functions	executeERC721LiquidationCall		

Description When an ERC721 is purchased during liquidation, any funds in excess of the selected debt to cover are supplied back to the user. If the user has more debt, these funds are marked as collateral by marking the reserve as collateral. The liquidated user might already have non-collateral funds in this reserve though, which means it is possible to mark additional funds as collateral against the user's intention. This could therefore cause tokens to be liquidated that the user had intended to protect from liquidation. Additionally, it is possible for liquidators to exploit this system so that they can gain access to more desirable tokens to liquidate in the future.

```
1    function executeERC721LiquidationCall(
2        mapping(address => DataTypes.ReserveData) storage reservesData,
3        mapping(uint256 => address) storage reservesList,
4        mapping(address => DataTypes.UserConfigurationMap) storage usersConfig,
5        DataTypes.ExecuteERC721LiquidationCallParams memory params
6    ) external {
7        ...
8
9        if (vars.collateralDiscountedPrice > toPrtocolAmount) {
10           if (vars.userGlobalTotalDebt > vars.actualDebtToLiquidate) {
11               SupplyLogic.executeSupply(reservesData, reservesList, userConfig,
12                   DataTypes.ExecuteSupplyParams({
13                       asset: params.liquidationAsset,
14                       amount: vars.collateralDiscountedPrice -
15                           vars.actualDebtToLiquidate -
16                           vars.liquidationProtocolFeeAmount,
17                       onBehalfOf: params.user,
18                       referralCode: 0 }));
19
20               if (!userConfig.isUsingAsCollateral(liquidationAssetReserveId)) {
21                   userConfig.setUsingAsCollateral(liquidationAssetReserveId, true);
22                   emit ReserveUsedAsCollateralEnabled(
23                       params.liquidationAsset,
24                       params.user);
25               }
26           }
27           ...
28       }
29       ...
30   }
```

Snippet 4.24: Location where the reserve is set as collateral after a supply

Recommendations This situation seems somewhat unlikely but users should be aware that this is a possibility.

Developer Response The developers acknowledged this issue and stated that they will consider possible design changes in future iterations to address this problem directly.

5 Other Recommendations

During the course of our audit, we noticed a number of optimization opportunities and inconsistencies in the code. While these issues do not constitute vulnerabilities, we include them in this report to guide further development efforts. For each issue found, we indicate the type of the issue, its location in the code, and its current status. Table 5.1 summarizes the issues found:

Table 5.1: Summary of Optimizations/Recommendations.

ID	Description	Status
V-PAR-INFO-001	Inconsistent Naming Convention	Acknowledged
V-PAR-INFO-002	Unconventional Casting	Acknowledged
V-PAR-INFO-003	Unnecessary Require in Loop	Fixed
V-PAR-INFO-004	Custom Errors	Acknowledged
V-PAR-INFO-005	Unused Variable/Computation	Fixed
V-PAR-INFO-006	Redundant check	Fixed

5.1 Results

In this section, we elaborate on the detailed description of each optimization/recommendation.

5.1.1 V-PAR-INFO-001: Inconsistent Naming Convention

Severity	Informational	**Commit**	bfc2d4b
Type	Code Consistency	**Status**	Acknowledged
Files	contracts/protocol/libraries/logic/ValidationLogic.sol		
Functions	validateDropReserve		

Description The code typically refers to an asset that can be either a PToken or an NToken as a xToken; however, some functions and variable names simply refer to a PToken even though they're intended to be used for both asset types.

```
    /**
     * @notice Updates the xToken implementation and initializes it
     * @dev Emits the 'PTokenUpgraded' event
     * @param cachedPool The Pool containing the reserve with the xToken
     * @param input The parameters needed for the initialize call
     */
    function executeUpdatePToken(
        IPool cachedPool,
        ConfiguratorInputTypes.UpdatePTokenInput calldata input
    ) public {
        DataTypes.ReserveData memory reserveData = cachedPool.getReserveData(
            input.asset
        );

        ...

        _upgradeTokenImplementation(
            reserveData.xTokenAddress,
            input.implementation,
            encodedCall
        );

        emit PTokenUpgraded(
            input.asset,
            reserveData.xTokenAddress,
            input.implementation
        );
    }
```

Snippet 5.1: Location where an NToken can be referred to as a PToken

Recommendations For clarity and consistency we believe the xToken naming convention should be adopted globally.

5.1.2 V-PAR-INFO-002: Unconventional Casting

Severity	Informational	**Commit**	bfc2d4b
Type	Code Consistency	**Status**	Acknowledged
Files	contracts/protocol/libraries/logic/ConfiguratorLogic.sol		
Functions	executeUpdatePToken		

Description There are several instances in which an address is cast to an interface type that the underlying contract does not inherit from. This is a dangerous pattern that can lead to maintenance problems since this makes it difficult to find all of the locations that need to be updated when a contract's interface changes. Major problems could arise from changing the interface but not updating a callsite, such as a denial of service. Note that in the below case, xTokenAddress can refer to an NToken which does not inherit from IERC20.

```
    function validateDropReserve(
        mapping(uint256 => address) storage reservesList,
        DataTypes.ReserveData storage reserve,
        address asset
    ) internal view {
        require(asset != address(0), Errors.ZERO_ADDRESS_NOT_VALID);
        require(
            reserve.id != 0 || reservesList[0] == asset,
            Errors.ASSET_NOT_LISTED
        );
        require(
            IERC20(reserve.stableDebtTokenAddress).totalSupply() == 0,
            Errors.STABLE_DEBT_NOT_ZERO
        );
        require(
            IERC20(reserve.variableDebtTokenAddress).totalSupply() == 0,
            Errors.VARIABLE_DEBT_SUPPLY_NOT_ZERO
        );
        require(
            IERC20(reserve.xTokenAddress).totalSupply() == 0,
            Errors.ATOKEN_SUPPLY_NOT_ZERO
        );
    }
```

Snippet 5.2: Location where an NToken can be cast to an IERC20 even though it doesn't inherit

Recommendations We recommend that calls to shared functions be made through shared interfaces.

5.1.3 V-PAR-INFO-003: Unnecessary Require in Loop

Severity	Informational	**Commit**	2226f6e
Type	Gas Optimization	**Status**	Fixed
Files	contracts/protocol/libraries/tokenization/MintableIncentivizedERC70		
Functions	_mintMultiple		

Description The require in the first statement of the for loop checks the variable "to" but it is never changed.

```
    function _mintMultiple(
        address to,
        DataTypes.ERC721SupplyParams[] calldata tokenData
    ) internal virtual returns (bool) {
        uint64 oldBalance = _userState[to].balance;
        uint256 oldTotalSupply = totalSupply();
        uint64 collaterizedTokens;

        for (uint256 index = 0; index < tokenData.length; index++) {
            require(to != address(0), "ERC721: mint to the zero address");
            ...
        }
        ...
    }
```

Snippet 5.3: Location where a require can be promoted outside of a loop

Recommendations Move this statement outside of the for loop.

5.1.4 V-PAR-INFO-004: Custom Errors

Severity	Informational	**Commit**	2226f6e
Type	Gas Optimization	**Status**	Acknowledged
Files		NA	
Functions		NA	

Description Long messages in a revert can consume large amounts of gas.

Recommendations We recommend the use of custom errors to cut down on gas requirements.

5.1.5 V-PAR-INFO-005: Unused Variable/Computation

Severity	Informational	Commit	19d718c
Type	Gas Optimization	Status	Fixed
Files	contracts/protocol/libraries/logic/SupplyLogic.sol		
Functions	executeSupplyERC721		

Description The number of collateralized tokens (usedAsCollateral) is calculated but never used.

```
1   function executeSupplyERC721(
2       mapping(address => DataTypes.ReserveData) storage reservesData,
3       mapping(uint256 => address) storage reservesList,
4       DataTypes.UserConfigurationMap storage userConfig,
5       DataTypes.ExecuteSupplyERC721Params memory params
6    ) external {
7       ...
8
9       uint256 usedAsCollateral;
10
11      for (uint256 index = 0; index < amount; index++) {
12          if (params.tokenData[index].useAsCollateral) {
13              usedAsCollateral++;
14          }
15
16          ...
17      }
18      ...
19  }
```

Snippet 5.4: Location where *usedAsCollateral* is computed

Recommendations If this variable isn't needed, remove it.

5.1.6 V-PAR-INFO-006: Redundant check

Severity	Informational	Commit	e747ba6
Type	Gas Optimization	Status	Fixed
Files	contracts/protocol/libraries/logic/ValidationLogic.sol		
Functions	validateERC721LiquidationCall		

Description The variable liquidatingERC721 is always true in function validateERC721LiquidationCall because of the earlier asset type check.

```
    function validateERC721LiquidationCall(
        DataTypes.UserConfigurationMap storage userConfig,
        DataTypes.ReserveData storage collateralReserve,
        DataTypes.ValidateERC721LiquidationCallParams memory params
    ) internal view {
        require(
            params.assetType == DataTypes.AssetType.ERC721,
            Errors.INVALID_ASSET_TYPE
        );

        ...

        bool liquidatingERC721 = params.assetType == DataTypes.AssetType.ERC721;

        if (liquidatingERC721) {
            require(
                params.liquidationAmount >= params.collateralDiscountedPrice,
                Errors.LIQUIDATION_AMOUNT_NOT_ENOUGH
            );
        }

        vars.isCollateralEnabled =
            collateralReserve.configuration.getLiquidationThreshold() != 0 &&
            userConfig.isUsingAsCollateral(collateralReserve.id) &&
            (
                liquidatingERC721
                    ? ICollaterizableERC721(params.xTokenAddress)
                        .isUsedAsCollateral(params.tokenId)
                    : true
            );

        ...
    }
```

Snippet 5.5: Location where a typecheck guarantees the value of *liquidatingERC721*

Recommendations Remove the liquidatingERC721 variable.