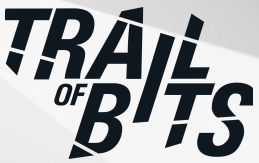

Paraspace

Fix Review

November 28, 2022

Prepared for:
Cheng Jiang
Ivan Solomonoff
Paraspace

Prepared by: **Will Song**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

Table of Contents

Executive Summary

Engagement Overview

Paraspace engaged Trail of Bits to review the security of its decentralized lending protocol. From October 3 to October 24, 2022, a team of three consultants conducted a security review of the client-provided source code, with seven person-weeks and two person-days of effort. Details of the project's scope, timeline, test targets, and coverage are provided in the original audit report.

Paraspace contracted Trail of Bits to review the fixes implemented for issues identified in the original report. On November 21, 2022, one consultant conducted a review of the client-provided source code, with one person-day of effort.

Summary of Findings

The original audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	0
Low	5
Informational	8
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	2
Auditing and Logging	1
Cryptography	1
Data Validation	3
Denial of Service	1
Error Reporting	1
Testing	1
Undefined Behavior	5

Overview of Fix Review Results

Of the 15 issues described in the original audit report, Paraspace has sufficiently addressed 6, partially resolved 1, and has accepted the risks associated with the 7 unfixed issues. Paraspace has claimed that one finding is not an issue in practice, but its actual status remains undetermined.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineers were associated with this project:

Will Song, Consultant
will.song@trailofbits.com

Tjaden Hess, Consultant
tjaden.hess@trailofbits.com

Samuel Moelius, Consultant
samuel.moelius@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
September 22, 2022	Pre-project kickoff call
October 11, 2022	Status update meeting #1
October 18, 2022	Status update meeting #2
October 25, 2022	Delivery of report draft
October 25, 2022	Final report readout
November 16, 2022	Delivery of final report
November 28, 2022	Delivery of fix review

Project Methodology

Our work in the fix review included the following:

- A review of the findings in the original audit report
- A manual review of the client-provided source code

Project Targets

The engagement involved a review of the fixes implemented in the targets listed

“Para-Space NFT Money Market”

Repository	https://github.com/para-space/paraspace-core
Versions	#69 #98 #109 #114 #139 #176
Type	Solidity
Platform	Ethereum

Summary of Fix Review Results

The table below summarizes each of the original findings and indicates whether the issue has been sufficiently resolved.

ID	Title	Status
1	Unconventional test structure	Unresolved
2	Insufficient event generation	Partially Resolved
3	Missing supportsInterface functions	Unresolved
4	ERC1155 asset type is defined but not implemented	Resolved
5	executeMintToTreasury silently skips non-ERC20 tokens	Unresolved
6	getReservesData does not set all AggregatedReserveData fields	Resolved
7	Excessive type repetition in returned tuples	Unresolved
8	Incorrect grace period could result in denial of service	Undetermined
9	Incorrect accounting in _transferCollaterizable	Unresolved
10	IPriceOracle interface is used only in tests	Unresolved
11	Manual ERC721 transfers could be claimed as NTokens by anyone	Resolved
12	Inconsistent behavior between NToken and PToken liquidations	Unresolved

13	Missing asset type checks in ValidationLogic library	**Resolved**
14	Uniswap v3 NFT flash claims may lead to undercollateralization	**Resolved**
15	Non-injective hash encoding in getClaimKeyHash	**Resolved**

Detailed Fix Review Results

1. Unconventional test structure

Status: **Unresolved**	
Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-PARASPACE-1
Target: `test-suites`	

Description
Aspects of the Paraspace tests make them difficult to run. Tests that are difficult to run are less likely to be run.

First, the Paraspace tests are configured to initialize all tests before any single test can be run. Therefore, even simple tests incur the initialization costs of the most expensive tests. Such a design hinders development.

Figure 1.1 shows the first 25 lines that are output during test initialization. Approximately 270 lines are output before the first test is run. As shown in the figure, several ERC20 and ERC721 tokens are deployed during initialization. These steps are unnecessary in many testing situations, such as if a user wants to run a test that does not involve these tokens.

```
- Environment
  - Network : hardhat
-> Deploying test environment...
------------ step 00 done ------------
deploying now  DAI
deploying now  WETH
deploying now  USDC
deploying now  USDT
deploying now  WBTC
deploying now  stETH
deploying now  APE
deploying now  aWETH
deploying now  cETH
deploying now  PUNK
------------ step 0A done ------------
deploying now  WPUNKS
deploying now  BAYC
deploying now  MAYC
```

```
deploying now  DOODLE
deploying now  AZUKI
deploying now  CLONEX
deploying now  MOONBIRD
deploying now  MEEBITS
deploying now  OTHR
deploying now  UniswapV3
...
```

Figure 1.1: The first 25 lines emitted by Paraspace tests

Second, the `paraspace-core` repository uses the `paraspace-deploy` repository as a Git submodule and relies on it when being built and tested. However, while the former is *public*, the latter is *private*. Therefore, `paraspace-core` can be built or tested only by those with access to `paraspace-deploy`.

Finally, some tests use nested `it` calls (figure 1.2), which are not supported by Mocha.

```
 it("deposited aWETH should have balance multiplied by rebasing index", async () =>
{
   ...
   it("should be able to supply aWETH and mint rebasing PToken", async () => {
     ...
   });

   it("expect the scaled balance to be the principal balance multiplied by Aave pool
liquidity index divided by RAY (2^27)", async () => {
     ...
   });
 });
```

Figure 1.2: `test-suites/rebasing.spec.ts#L125–L165`

Developers should strive to implement testing that thoroughly covers the project and tests against both bad and expected inputs. Having robust unit and integration tests can greatly increase both developers' and users' confidence in the code's functionality. However, tests cannot benefit the system if they are not actually run. Therefore, tests should be made as easy to run as possible.

Fix Analysis

The issue is unresolved. The Paraspace team has determined that this issue does not need to be urgently addressed before the mainnet launch.

2. Insufficient event generation	
Status: **Partially Resolved**	
Severity: **Low**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-PARASPACE-2
Target: Various targets	

Description
Events generated during contract execution aid in monitoring, baselining of behavior, and detection of suspicious activity. Without events, users and blockchain-monitoring systems cannot easily detect behavior that falls outside the baseline conditions. Consequently, malfunctioning contracts or attacks may not be detected.

Multiple critical operations do not emit events. As a result, it will be difficult to review the correct behavior of the contracts once they have been deployed.

Generally speaking, an operation should emit an event if it involves any of the following:

- A transfer of an asset
- A change to a contract parameter
- Privileged roles

Moreover, it is not always sufficient to rely on events emitted by subordinate operations. For example, the following `emergencyTokenTransfer` operations should emit their own specific events:

- `MoonBirdsGateway.emergencyTokenTransfer`
- `WETHGateway.emergencyTokenTransfer`
- `UniswapV3Gateway.emergencyTokenTransfer`
- `WPunkGateway.emergencyTokenTransfer`

In addition to the above, the following events are defined but never emitted. We recommend reviewing this list to determine whether the events should be emitted.

- `AggregatorInterface.AnswerUpdated`

- `AggregatorInterface.NewRound`
- `IEACAggregatorProxy.AnswerUpdated`
- `IEACAggregatorProxy.AnswerUpdated`
- `IEACAggregatorProxy.NewRound`
- `IEACAggregatorProxy.NewRound`
- `INonfungiblePositionManager.DecreaseLiquidity`
- `INonfungiblePositionManager.IncreaseLiquidity`
- `IRewardController.ClaimerSet`
- `IRewardController.RewardsAccrued`
- `IRewardController.RewardsClaimed`
- `IRewardController.RewardsClaimed`
- `IRewardsController.ClaimerSet`
- `IRewardsController.RewardOracleUpdated`
- `IRewardsController.RewardsClaimed`
- `IRewardsController.TransferStrategyInstalled`
- `IRewardsDistributor.Accrued`
- `IRewardsDistributor.AssetConfigUpdated`
- `IRewardsDistributor.EmissionManagerUpdated`
- `ITransferStrategyBase.EmergencyWithdrawal`

Fix Analysis

The issue has been partially resolved. Through pull request #176, the Paraspace team added events to be emitted by `WETHGateway.emergencyTokenTransfer` and `WPunkGateway.emergencyERC721TokenTransfer`. The other two methods that were missing events were removed in a previous update to the codebase. However, the events that are defined but not emitted remain unchanged.

3. Missing supportsInterface functions

Status: **Unresolved**	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PARASPACE-3
Target: Various contracts	

Description

According to EIP-165, a contract's implementation of the `supportsInterface` function should return `true` for the interfaces that the contract supports. Outside of the dependencies and mocks directories, only one Paraspace contract has a `supportsInterface` function.

For example, each of the following contracts includes an `onERC721Received` function; therefore, they should have a `supportsInterface` function that returns `true` for the `ERC721TokenReceiver` interface (`PoolCore`'s `onERC721Received` implementation appears in figure 3.1):

- `contracts/ui/MoonBirdsGateway.sol`
- `contracts/ui/UniswapV3Gateway.sol`
- `contracts/ui/WPunkGateway.sol`
- `contracts/protocol/tokenization/NToken.sol`
- `contracts/protocol/tokenization/NTokenUniswapV3.sol`
- `contracts/protocol/tokenization/NTokenMoonBirds.sol`
- `contracts/protocol/pool/PoolCore.sol`

```
// This function is necessary when receive erc721 from looksrare
function onERC721Received(
    address,
    address,
    uint256,
    bytes memory
) external virtual returns (bytes4) {
    return this.onERC721Received.selector;
```

```
}
```

Figure 3.1: contracts/protocol/pool/PoolCore.sol#L773–L781

Fix Analysis

The issue is unresolved. The Paraspace team accepts the risk associated with this finding.

4. ERC1155 asset type is defined but not implemented

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PARASPACE-4
Target: `contracts/protocol/libraries/{logic/PoolLogic.sol, types/DataTypes.sol}`	

Description

The asset type `ERC1155` is defined in `DataTypes.sol` but is not otherwise supported. Having an unsupported variant in the code is risky, as developers could use it accidentally.

The `AssetType` declaration appears in figure 4.1. It consists of three variants, one of which is `ERC1155`. However, `ERC1155` does not appear anywhere else in the code. For example, it does not appear in the `executeRescueTokens` function in the `PoolLogic.sol` contract (figure 4.2), meaning it is not possible to rescue ERC1155 tokens.

```
enum AssetType {
    ERC20,
    ERC721,
    ERC1155
}
```

Figure 4.1: contracts/protocol/libraries/types/DataTypes.sol#L7–L11

```
function executeRescueTokens(
    DataTypes.AssetType assetType,
    address token,
    address to,
    uint256 amountOrTokenId
) external {
    if (assetType == DataTypes.AssetType.ERC20) {
        IERC20(token).safeTransfer(to, amountOrTokenId);
    } else if (assetType == DataTypes.AssetType.ERC721) {
        IERC721(token).safeTransferFrom(address(this), to, amountOrTokenId);
    }
}
```

Figure 4.2: contracts/protocol/libraries/logic/PoolLogic.sol#L80–L91

Fix Analysis

The issue has been resolved. Through pull request #114, the Paraspace team has removed the ERC1155 asset type.

5. executeMintToTreasury silently skips non-ERC20 tokens

Status: **Unresolved**	
Severity: **Low**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-PARASPACE-5
Target: `contracts/protocol/{libraries/logic/PoolLogic.sol, pool/PoolParameters.sol}`	

Description

The `executeMintToTreasury` function silently ignores non-ERC20 assets passed to it. Such behavior could allow erroneous calls to `executeMintToTreasury` to go unnoticed.

The code for `executeMintToTreasury` appears in figure 5.1. It is called from the `mintToTreasury` function in `PoolParameters.sol` (figure 5.2). As shown in figure 5.1, non-ERC20 assets are silently skipped.

```
function executeMintToTreasury(
    mapping(address => DataTypes.ReserveData) storage reservesData,
    address[] calldata assets
) external {
    for (uint256 i = 0; i < assets.length; i++) {
        address assetAddress = assets[i];

        DataTypes.ReserveData storage reserve = reservesData[assetAddress];

        DataTypes.ReserveConfigurationMap
            memory reserveConfiguration = reserve.configuration;

        // this cover both inactive reserves and invalid reserves since the flag
will be 0 for both
        if (
            !reserveConfiguration.getActive() ||
            reserveConfiguration.getAssetType() != DataTypes.AssetType.ERC20
        ) {
            continue;
        }
        ...
    }
}
```

Figure 5.1: contracts/protocol/libraries/logic/PoolLogic.sol#L98–L134

```
function mintToTreasury(address[] calldata assets)
    external
    virtual
    override
    nonReentrant
{
    PoolLogic.executeMintToTreasury(_reserves, assets);
}
```

Figure 5.2: contracts/protocol/pool/PoolParameters.sol#L97–L104

Note that because this is a minting operation, it likely meant to be called by an administrator. However, an administrator could pass a non-ERC20 asset in error. Because the function silently skips such assets, the error could go unnoticed.

Fix Analysis

The issue is unresolved. The Paraspace team accepts the risk associated with this finding.

6. getReservesData does not set all AggregatedReserveData fields

Status: **Resolved**	
Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PARASPACE-6
Target: `contracts/ui/{interfaces/IUiPoolDataProvider.sol, UiPoolDataProvider.sol}`	

Description

The `getReservesData` function fills in an `AggregatedReserveData` structure for the reserve handled by an `IPoolAddressesProvider`. However, the function does not set the structure's `name` and `assetType` fields. Therefore, off-chain code relying on this function will see uninitialized data.

Part of the `AggregatedReserveData` structure appears in figure 6.1. The complete structure consists of 53 fields. Each iteration of the loop in `getReservesData` (figure 6.2) fills in the fields of one `AggregatedReserveData` structure. However, the loop does not set the structures' `name` fields. And although reserve `assetTypes` are computed, they are never stored in the structure.

```
struct AggregatedReserveData {
    address underlyingAsset;
    string name;
    string symbol;
    ...
    //AssetType
    DataTypes.AssetType assetType;
}
```

Figure 6.1: contracts/ui/interfaces/IUiPoolDataProvider.sol#L18–L78

```
function getReservesData(IPoolAddressesProvider provider)
    public
    view
    override
    returns (AggregatedReserveData[] memory, BaseCurrencyInfo memory)
{
    IParaSpaceOracle oracle = IParaSpaceOracle(provider.getPriceOracle());
    IPool pool = IPool(provider.getPool());

    address[] memory reserves = pool.getReservesList();
```

```
    AggregatedReserveData[]
        memory reservesData = new AggregatedReserveData[](reserves.length);

    for (uint256 i = 0; i < reserves.length; i++) {
        ...
        DataTypes.AssetType assetType;
        (
            reserveData.isActive,
            reserveData.isFrozen,
            reserveData.borrowingEnabled,
            reserveData.stableBorrowRateEnabled,
            isPaused,
            assetType
        ) = reserveConfigurationMap.getFlags();
        ...
    }
    ...
    return (reservesData, baseCurrencyInfo);
}
```

Figure 6.2: contracts/ui/UiPoolDataProvider.sol#L83–L269

Fix Analysis

The issue has been resolved. Through pull requests #176 and #69, the Paraspace team has adjusted the `getReservesData` function so that it sets the `name` and `assetType` fields, respectively.

7. Excessive type repetition in returned tuples

Status: **Unresolved**	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PARASPACE-7
Target: `contracts/protocol/libraries/{logic/GenericLogic.sol, configuration/ReserveConfiguration.sol}`	

Description

Several functions return tuples that contain many fields of the same type adjacent to one another. Such a practice is error-prone, as callers could easily confuse the fields.

An example appears in figure 7.1. The tuple returned by the `calculateUserAccountData` function contains nine fields of type `uint256` adjacent to each other. An example in which the function is called appears in figure 7.2. As the figure makes evident, a misplaced comma, indicating that the caller identified the wrong field holding the data of interest, could have disastrous consequences.

```
/**
 * @notice Calculates the user data across the reserves.
 * @dev It includes the total liquidity/collateral/borrow balances in the base
currency used by the price feed,
 * the average Loan To Value, the average Liquidation Ratio, and the Health factor.
 * @param reservesData The state of all the reserves
 * @param reservesList The addresses of all the active reserves
 * @param params Additional parameters needed for the calculation
 * @return The total collateral of the user in the base currency used by the price
feed
 * @return The total ERC721 collateral of the user in the base currency used by the
price feed
 * @return The total debt of the user in the base currency used by the price feed
 * @return The average ltv of the user
 * @return The average liquidation threshold of the user
 * @return The health factor of the user
 * @return True if the ltv is zero, false otherwise
 **/
function calculateUserAccountData(
    mapping(address => DataTypes.ReserveData) storage reservesData,
    mapping(uint256 => address) storage reservesList,
    DataTypes.CalculateUserAccountDataParams memory params
)
    internal
```

```
    view
    returns (
        uint256,
        uint256,
        uint256,
        uint256,
        uint256,
        uint256,
        uint256,
        uint256,
        uint256,
        bool
    )
{
    ...
    return (
        vars.totalCollateralInBaseCurrency,
        vars.totalERC721CollateralInBaseCurrency,
        vars.totalDebtInBaseCurrency,
        vars.avgLtv,
        vars.avgLiquidationThreshold,
        vars.avgERC721LiquidationThreshold,
        vars.payableDebtByERC20Assets,
        vars.healthFactor,
        vars.erc721HealthFactor,
        vars.hasZeroLtvCollateral
    );
}
```

Figure 7.1: contracts/protocol/libraries/logic/GenericLogic.sol#L58–L302

```
(
    vars.userGlobalCollateralBalance,
    ,
    vars.userGlobalTotalDebt,
    ,
    ,
    ,
    ,
    ,
    vars.healthFactor,

) = GenericLogic.calculateUserAccountData(
```

Figure 7.2: contracts/protocol/libraries/logic/LiquidationLogic.sol#L393–L404

Also, note that the documentation of `calculateUserAccountData` does not accurately reflect the implementation. The documentation describes only six returned `uint256` fields (highlighted in yellow in figure 7.1). In reality, the function returns an additional three (highlighted in red in figure 7.1).

Less extreme but similar examples of adjacent field types in tuples appear in figures 7.3 and 7.4.

```
function getFlags(DataTypes.ReserveConfigurationMap memory self)
    internal
    pure
    returns (
        bool,
        bool,
        bool,
        bool,
        bool,
        DataTypes.AssetType
    )
```

Figure 7.3: contracts/protocol/libraries/configuration/ReserveConfiguration.sol#L516–L526

```
function getParams(DataTypes.ReserveConfigurationMap memory self)
    internal
    pure
    returns (
        uint256,
        uint256,
        uint256,
        uint256,
        uint256,
        bool
    )
```

Figure 7.4: contracts/protocol/libraries/configuration/ReserveConfiguration.sol#L552–L562

Fix Analysis

The issue is unresolved. The Paraspace team accepts the risk associated with this finding.

8. Incorrect grace period could result in denial of service

Status: **Undetermined**	
Severity: **Low**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-PARASPACE-8
Target: `contracts/protocol/configuration/PriceOracleSentinel.sol`	

Description

The `PriceOracleSentinel` contract's `isBorrowAllowed` and `isLiquidationAllowed` functions return `true` only if a "grace period" has elapsed since the oracle's last update. Setting the grace period parameter too high could result in a denial-of-service condition.

The relevant code appears in figure 8.1. Both `isBorrowAllowed` and `isLiquidationAllowed` call `_isUpAndGracePeriodPassed`, which checks whether `block.timestamp` minus `lastUpdateTimestamp` is greater than `_gracePeriod`.

```
/// @inheritdoc IPriceOracleSentinel
function isBorrowAllowed() external view override returns (bool) {
    return _isUpAndGracePeriodPassed();
}

/// @inheritdoc IPriceOracleSentinel
function isLiquidationAllowed() external view override returns (bool) {
    return _isUpAndGracePeriodPassed();
}

/**
 * @notice Checks the sequencer oracle is healthy: is up and grace period passed.
 * @return True if the SequencerOracle is up and the grace period passed, false
otherwise
 */
function _isUpAndGracePeriodPassed() internal view returns (bool) {
    (, int256 answer, , uint256 lastUpdateTimestamp, ) = _sequencerOracle
        .latestRoundData();
    return
        answer == 0 && block.timestamp - lastUpdateTimestamp > _gracePeriod;
}
```

Figure 8.1: contracts/protocol/configuration/PriceOracleSentinel.sol#L69–L88

Suppose `block.timestamp` minus `lastUpdateTimestamp` is never more than N seconds. Consequently, setting `_gracePeriod` to N or greater would mean that `isBorrowAllowed` and `isLiquidationAllowed` never return `true`.

The code in figure 8.1 resembles some example code from the Chainlink documentation. However, in that example code, the "grace period" is relative to when the round started, not when the round was updated.

Fix Analysis

The fix status of the issue is undetermined. The Paraspace team has not implemented any fixes for this finding, as it has determined that the denial of service will not occur in practice. However, we have not investigated this claim.

9. Incorrect accounting in _transferCollaterizable

Status: **Unresolved**	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PARASPACE-9
Target: `contracts/protocol/tokenization/{NToken.sol, base/MintableIncentivizedERC721.sol}`, `test-suites/ntoken.spec.ts`	

Description

The `_transferCollaterizable` function mishandles the `collaterizedBalance` and `_isUsedAsCollateral` fields. At a minimum, this means that transferred tokens cannot be used as collateral.

The code for `_transferCollaterizable` appears in figure 9.1. It is called from `Ntoken._transfer` (figure 9.2). The code decreases `_userState[from].collaterizedBalance` and clears `_isUsedAsCollateral[tokenId]`. However, the code does not make any corresponding changes, such as increasing `_userState[to].collaterizedBalance` and setting `_isUsedAsCollateral[tokenId]` elsewhere. As a result, if Alice transfers her NToken to Bob, Bob will not be able to use the corresponding ERC721 token as collateral.

```
function _transferCollaterizable(
    address from,
    address to,
    uint256 tokenId
) internal virtual returns (bool isUsedAsCollateral_) {
    isUsedAsCollateral_ = _isUsedAsCollateral[tokenId];

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

    MintableIncentivizedERC721._transfer(from, to, tokenId);
}
```

Figure 9.1: contracts/protocol/tokenization/base/MintableIncentivizedERC721.sol#L643–L656

```
function _transfer(
```

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

    uint256 fromBalanceBefore = collaterizedBalanceOf(from);
    uint256 toBalanceBefore = collaterizedBalanceOf(to);
    bool isUsedAsCollateral = _transferCollaterizable(from, to, tokenId);
    ...
}
```

Figure 9.2: contracts/protocol/tokenization/NToken.sol#L300–L324

The code used to verify the bug appears in figure 9.3. The code first verifies that the `collaterizedBalance` and `_isUsedAsCollateral` fields are set correctly. It then has User 1 send his or her token to User 2, who sends it back to User 1. Finally, it verifies that the `collaterizedBalance` and `_isUsedAsCollateral` fields are set *incorrectly*. Most subsequent tests fail thereafter.

```
it("User 1 sends the nToken to User 2, who sends it back to User 1", async () => {
  const {
    nBAYC,
    users: [user1, user2],
  } = testEnv;

  expect(await nBAYC.isUsedAsCollateral(0)).to.be.equal(true);
  expect(await nBAYC.collaterizedBalanceOf(user1.address)).to.be.equal(1);
  expect(await nBAYC.collaterizedBalanceOf(user2.address)).to.be.equal(0);

  await nBAYC.connect(user1.signer).transferFrom(user1.address, user2.address, 0);

  await nBAYC.connect(user2.signer).transferFrom(user2.address, user1.address, 0);

  expect(await nBAYC.isUsedAsCollateral(0)).to.be.equal(false);
  expect(await nBAYC.collaterizedBalanceOf(user1.address)).to.be.equal(0);
  expect(await nBAYC.collaterizedBalanceOf(user2.address)).to.be.equal(0);
});

it("User 2 deposits 10k DAI and User 1 borrows 8K DAI", async () => {
```

Figure 9.3: This is the code used to verify the bug. The highlighted line appears in the `ntoken.spec.ts` file. What precedes it was added to that file.

Fix Analysis

The issue is unresolved. The Paraspace team accepts the risk associated with this finding.

10. IPriceOracle interface is used only in tests

Status: **Unresolved**	
Severity: **Informational**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-PARASPACE-10
Target: `contracts/interfaces/IPriceOracle.sol`	

Description

The `IPriceOracle` interface is used only in tests, yet it appears alongside production code. Its location increases the risk that a developer will try to use it in production code.

The complete interface appears in figure 10.1. Note that the interface includes code that a real oracle is unlikely to include, such as the `setAssetPrice` function. Therefore, a developer that calls this function would likely introduce a bug into the code.

```
// SPDX-License-Identifier: AGPL-3.0
pragma solidity 0.8.10;

/**
 * @title IPriceOracle
 *
 * @notice Defines the basic interface for a Price oracle.
 **/
interface IPriceOracle {
    /**
     * @notice Returns the asset price in the base currency
     * @param asset The address of the asset
     * @return The price of the asset
     **/
    function getAssetPrice(address asset) external view returns (uint256);

    /**
     * @notice Set the price of the asset
     * @param asset The address of the asset
     * @param price The price of the asset
     **/
    function setAssetPrice(address asset, uint256 price) external;
}
```

Figure 10.1: `contracts/interfaces/IPriceOracle.sol`

Fix Analysis

The issue is unresolved. The Paraspace team accepts the risk associated with this finding.

11. Manual ERC721 transfers could be claimed as NTokens by anyone

Status: **Resolved**	
Severity: **High**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-PARASPACE-11
Target: `contracts/protocol/tokenization/NToken.sol`	

Description

The `PoolCore` contract has an external function `supplyERC721FromNToken`, whose purpose is to validate that the given ERC721 assets are owned by the `NToken` contract and then to mint the corresponding NTokens to a caller-supplied address. We suspect that the intended use case for this function is that the `NTokenMoonBirds` or `UniswapV3Gateway` contract will transfer the ERC721 assets to the `NToken` contract and then immediately call `supplyERC721FromNToken`. However, the access controls on this function allow an unauthorized user to take ownership of any assets manually transferred to the `NToken` contract, for whatever reason that may be, as `NToken` does not track the original owner of the asset.

```
function supplyERC721FromNToken(
    address asset,
    DataTypes.ERC721SupplyParams[] calldata tokenData,
    address onBehalfOf
) external virtual override nonReentrant {
    SupplyLogic.executeSupplyERC721FromNToken(
        // ...
    );
}
```

Figure 11.1: The external `supplyERC721FromNToken` function within `PoolCore`

```
function validateSupplyFromNToken(
    DataTypes.ReserveCache memory reserveCache,
    DataTypes.ExecuteSupplyERC721Params memory params,
    DataTypes.AssetType assetType
) internal view {
    // ...
    for (uint256 index = 0; index < amount; index++) {
        // validate that the owner of the underlying asset is the NToken  contract
        require(
            IERC721(params.asset).ownerOf(
                params.tokenData[index].tokenId
```

```
            ) == reserveCache.xTokenAddress,
            Errors.NOT_THE_OWNER
        );
        // validate that the owner of the ntoken that has the same tokenId is the
zero address
        require(
            IERC721(reserveCache.xTokenAddress).ownerOf(
                params.tokenData[index].tokenId
            ) == address(0x0),
            Errors.NOT_THE_OWNER
        );
    }
}
```

Figure 11.2: The validation checks performed by `supplyERC721FromNToken`

```
function executeSupplyERC721Base(
    uint16 reserveId,
    address nTokenAddress,
    DataTypes.UserConfigurationMap storage userConfig,
    DataTypes.ExecuteSupplyERC721Params memory params
) internal {
    // ...
    bool isFirstCollaterarized = INToken(nTokenAddress).mint(
        params.onBehalfOf,
        params.tokenData
    );
    // ...
}
```

Figure 11.3: The unauthorized minting operation

Users regularly interact with the `NToken` contract, which represents ERC721 assets, so it is possible that a malicious actor could convince users to transfer their ERC721 assets to the contract in an unintended manner.

Fix Analysis

The issue is resolved. Through pull request #98, the Paraspace team has added a check in the validation logic to ensure that only the XToken contract is allowed to call `SupplyLogic.executeSupplyERC721FromNToken`.

12. Inconsistent behavior between NToken and PToken liquidations

Status: **Unresolved**	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PARASPACE-12
Target: `contracts/protocol/libraries/logic/LiquidationLogic.sol`	

Description

When a user liquidates another user's ERC20 tokens and opts to receive PTokens, the PTokens are automatically registered as collateral. However, when a user liquidates another user's ERC721 token and opts to receive an NToken, the NToken is not automatically registered as collateral. This discrepancy could be confusing for users.

The relevant code appears in figures 12.1 through 12.3. For ERC20 tokens, `_liquidatePTokens` is called, which in turns calls `setUsingAsCollateral` if the liquidator has not already designated the PTokens as collateral (figures 12.1 and 12.2). However, for an ERC721 token, the NToken is simply transferred (figure 12.3).

```
if (params.receiveXToken) {
    _liquidatePTokens(usersConfig, collateralReserve, params, vars);
} else {
```

Figure 12.1:
contracts/protocol/libraries/logic/LiquidationLogic.sol#L310–L312

```
function _liquidatePTokens(
    mapping(address => DataTypes.UserConfigurationMap) storage usersConfig,
    DataTypes.ReserveData storage collateralReserve,
    DataTypes.ExecuteLiquidationCallParams memory params,
    LiquidationCallLocalVars memory vars
) internal {
    ...
    if (liquidatorPreviousPTokenBalance == 0) {
        DataTypes.UserConfigurationMap
            storage liquidatorConfig = usersConfig[vars.liquidator];

        liquidatorConfig.setUsingAsCollateral(collateralReserve.id, true);
        emit ReserveUsedAsCollateralEnabled(
            params.collateralAsset,
            vars.liquidator
        );
    }
```

```
}
```

Figure 12.2: contracts/protocol/libraries/logic/LiquidationLogic.sol#L667–L693

```
if (params.receiveXToken) {
    INToken(vars.collateralXToken).transferOnLiquidation(
        params.user,
        vars.liquidator,
        params.collateralTokenId
    );
} else {
```

Figure 12.3: contracts/protocol/libraries/logic/LiquidationLogic.sol#L562–L568

Fix Analysis

The issue is unresolved. The Paraspace team accepts the risk associated with this finding.

13. Missing asset type checks in ValidationLogic library

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PARASPACE-13
Target: `contracts/protocol/libraries/logic/ValidationLogic.sol`	

Description

Some validation functions involving assets do not check the given asset's type. Such checks should be added to ensure defense in depth.

The `validateRepay` function is one example (figure 13.1). The function performs several checks involving the asset being repaid, but the function does not check that the asset is an ERC20 asset.

```
function validateRepay(
    DataTypes.ReserveCache memory reserveCache,
    uint256 amountSent,
    DataTypes.InterestRateMode interestRateMode,
    address onBehalfOf,
    uint256 stableDebt,
    uint256 variableDebt
) internal view {
    ...
    (bool isActive, , , , bool isPaused, ) = reserveCache
        .reserveConfiguration
        .getFlags();
    require(isActive, Errors.RESERVE_INACTIVE);
    require(!isPaused, Errors.RESERVE_PAUSED);
    ...
}
```

Figure 13.1: contracts/protocol/libraries/logic/ValidationLogic.sol#L403–L447

Another example is the `validateFlashloanSimple` function, which does not check that the loaned asset is an ERC20 asset.

We do not believe that the absence of these checks currently represents a vulnerability. However, adding these checks will help protect the code against future modifications.

Fix Analysis

The issue has been resolved. Through pull request #176, the Paraspace team has added several asset type checks to the core validation functions where needed.

14. Uniswap v3 NFT flash claims may lead to undercollateralization

Status: **Resolved**	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PARASPACE-14
Target: `contracts/protocol/libraries/logic/FlashClaimLogic.sol`	

Description

Flash claims enable users with collateralized NFTs to assume ownership of the underlying asset for the duration of a single transaction, with the condition that the NFT be returned at the end of the transaction. When used with typical NFTs, such as Bored Ape Yacht Club tokens, the atomic nature of flash claims prevents users from removing net value from the Paraspace contract while enabling them to claim rewards, such as airdrops, that they are entitled to by virtue of owning the NFTs.

Uniswap v3 NFTs represent a position in a Uniswap liquidity pool and entitle the owner to add or withdraw liquidity from the underlying Uniswap position. Uniswap v3 NFT prices are determined by summing the value of the two ERC20 tokens deposited as liquidity in the underlying position. Normally, when a Uniswap NFT is deposited in the Uniswap `NToken` contract, the user can withdraw liquidity only if the resulting price leaves the user's health factor above one. However, by leveraging the flash claim system, a user could claim the Uniswap v3 NFT temporarily and withdraw liquidity directly, returning a valueless NFT.

As currently implemented, Paraspace is not vulnerable to this attack because Uniswap v3 flash claims are, apparently accidentally, nonfunctional. A check in the `onERC721Recieved` function of the `NTokenUniswapV3` contract, which is designed to prevent users from depositing Uniswap positions via the `supplyERC721` method, incidentally prevents Uniswap NFTs from being returned to the contract during the flash claim process. However, this check could be removed in future updates and occurs at the very last step in what would otherwise be a successful exploit.

```
function onERC721Received(
    address operator,
    address,
    uint256 id,
    bytes memory
) external virtual override returns (bytes4) {

    // ...
```

```
    // if the operator is the pool, this means that the pool is transferring the
token to this contract
    // which can happen during a normal supplyERC721 pool tx
    if (operator == address(POOL)) {
        revert(Errors.OPERATION_NOT_SUPPORTED);
    }
```

Figure 14.1: The failing check that prevents the completion of Uniswap v3 flash claims

Fix Analysis

The issue has been resolved. Through pull request #109, the Paraspace team has disallowed `NTokenUniswapV3` flash claims by preventing the validation from succeeding.

15. Non-injective hash encoding in getClaimKeyHash

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-PARASPACE-15
Target: `contracts/misc/flashclaim/AirdropFlashClaimReceiver.sol`	

Description

As part of the flash claim functionality, Paraspace provides an implementation of a contract that can claim airdrops on behalf of NFT holders. This contract tracks claimed airdrops in the `airdropClaimRecords` mapping, indexed by the result of the `getClaimKeyHash` function. However, it is possible for two different inputs to `getClaimKeyHash` to result in identical hashes through a collision in the unpacked encoding. Because `nftTokenIds` and `params` are both variable-length inputs, an input with `nftTokenIds` equal to `uint256(1)` and an empty `params` will hash to the same value as an input with an empty `nftTokenIds` and `params` equal to `uint256(1)`.

Although the `airdropClaimRecords` mapping is not read or otherwise referenced elsewhere in the code, collisions may cause off-chain clients to mistakenly believe that an unclaimed airdrop has already been claimed.

```
function getClaimKeyHash(
    address initiator,
    address nftAsset,
    uint256[] calldata nftTokenIds,
    bytes calldata params
) public pure returns (bytes32) {
    return
        keccak256(
            abi.encodePacked(initiator, nftAsset, nftTokenIds, params)
        );
}
```

Figure 15.1: contracts/misc/flashclaim/AirdropFlashClaimReceiver.sol#L247–257

Fix Analysis

The issue has been resolved. Through pull request #139, the Paraspace team has changed the call to `getClaimKeyHash` to utilize `abi.encode` rather than `abi.encodePacked`.

A. Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

B. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.