SHERLOCK

SHERLOCK SECURITY REVIEW FOR

SHERLOCK

Prepared for: Astaria
Prepared by: Sherlock
Lead Security Expert: 0xRajeev
Dates Audited: October 20 - November 3, 2022
Prepared on: November 17, 2022

Introduction

Astaria's mission is to build a highly functional on-chain lending protocol, with instant highly liquid NFT lending.

Scope

```
./lib/astaria-gpl/src/ERC4626-Cloned.sol
./lib/astaria-gpl/src/ERC721.sol
./lib/astaria-gpl/src/AuctionHouse.sol
./src/strategies/UniqueValidator.sol
./src/strategies/CollectionValidator.sol
./src/strategies/UNI_V3Validator.sol
./src/PublicVault.sol
./src/LiquidationAccountant.sol
./src/AstariaRouter.sol
./src/TransferProxy.sol
./src/VaultImplementation.sol
./src/LienToken.sol
./src/CollateralToken.sol
./src/security/V3SecurityHook.sol
./src/WithdrawProxy.sol
```

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
44	35

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

0xRajeev
obront
rvierdiiev
ctf_sec
bin2chen
Jeiwan
TurnipBoy
csanuragjain
__141345__
8olidity
joestakey
Bnke0x0
hansfriese
neila
0x4141
Prefix
sorrynotsorry
yixxas
zzykxx
0x0
HonorLt
cryptphi
0xNazgul
tives
minhquanym
peanuts
chainNue
ak1
pashov
supernova
cccz
seyni
Nyx
Sm4rty
Rohan16

Issue H-1: The implied value of a public vault can be impaired, liquidity providers can lose funds

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/272

Found by

Jeiwan

Summary

The implied value of a public vault can be impaired, liquidity providers can lose funds

Vulnerability Detail

Borrowers can partially repay their liens, which is handled by the `_payment` function (LienToken.sol#L594). When repaying a part of a lien, `lien.amount` is updated to include currently accrued debt (LienToken.sol#L605-L617):

```
Lien storage lien = lienData[lienId];
lien.amount = _getOwed(lien); // @audit current debt, including accrued interest;
↪  saved to storage!
```

Notice that `lien.amount` is updated in storage, and `lien.last` wasn't updated.

Then, lien's slope is subtracted from vault's slope accumulator to be re-calculated after the repayment (LienToken.sol#L620-L630):

```
if (isPublicVault) {
  // @audit calculates and subtracts lien's slope from vault's slope
  IPublicVault(lienOwner).beforePayment(lienId, paymentAmount);
}
if (lien.amount > paymentAmount) {
  lien.amount -= paymentAmount;
  // @audit lien.last is updated only after payment amount subtraction
  lien.last = block.timestamp.safeCastTo32();
  // slope does not need to be updated if paying off the rest, since we
↪  neutralize slope in beforePayment()
  if (isPublicVault) {
    // @audit re-calculates and re-applies lien's slope after the repayment
    IPublicVault(lienOwner).afterPayment(lienId);
  }
}
```

In the `beforePayment` function, `LIEN_TOKEN().calculateSlope(lienId)` is called to calculate lien's current slope (PublicVault.sol#L433-L442):

```
function beforePayment(uint256 lienId, uint256 amount) public onlyLienToken {
  _handleStrategistInterestReward(lienId, amount);
  uint256 lienSlope = LIEN_TOKEN().calculateSlope(lienId);
  if (lienSlope > slope) {
    slope = 0;
  } else {
    slope -= lienSlope;
  }
  last = block.timestamp;
}
```

The `calculateSlope` function reads a lien from storage and calls `_getOwed` again (LienToken.sol#L440-L445):

```
function calculateSlope(uint256 lienId) public view returns (uint256) {
  // @audit lien.amount includes interest accrued so far
  Lien memory lien = lienData[lienId];
  uint256 end = (lien.start + lien.duration);
  uint256 owedAtEnd = _getOwed(lien, end);
  // @audit lien.last wasn't updated in `_payment`, it's an older timestamp
  return (owedAtEnd - lien.amount).mulDivDown(1, end - lien.last);
}
```

This is where double counting of accrued interest happens. Recall that lien's amount already includes the interest that was accrued by this moment (in the `_payment` function). Now, interest is calculated again and *is applied to the amount that already includes (a portion) it* (LienToken.sol#L544-L550):

```
function _getOwed(Lien memory lien, uint256 timestamp)
  internal
  view
  returns (uint256)
{
  // @audit lien.amount already includes interest accrued so far
  return lien.amount + _getInterest(lien, timestamp);
}
```

LienToken.sol#L177-L196:

```
function _getInterest(Lien memory lien, uint256 timestamp)
  internal
  view
  returns (uint256)
{
  if (!lien.active) {
    return uint256(0);
```

```
  }
  uint256 delta_t;
  if (block.timestamp >= lien.start + lien.duration) {
    delta_t = uint256(lien.start + lien.duration - lien.last);
  } else {
    // @audit lien.last wasn't updated in `_payment`, so the `delta_t` is bigger
↪   here
    delta_t = uint256(timestamp.safeCastTo32() - lien.last);
  }
  return
    // @audit rate applied to a longer delta_t and multiplied by a bigger amount
↪   than expected
    delta_t.mulDivDown(lien.rate, 1).mulDivDown(
      lien.amount,
      INTEREST_DENOMINATOR
    );
}
```

Impact

Double counting of interest will result in a wrong lien slope, which will affect the vault's slope accumulator. This will result in an invalid implied value of a vault (PublicVault.sol#L406-L413):

1. If miscalculated lien slope is bigger than expected, vault's slope will be smaller than expected (due to the subtraction in `beforePayment`), and vault's implied value will also be smaller. Liquidity providers will lose money because they won't be able to redeem the whole liquidity (vault's implied value, `totalAssets`, is used in the conversion of LP shares, ERC4626-Cloned.sol#L392-L412)
2. If miscalculated lien slope is smaller than expected, vault's slope will be higher, and vaults implied value will also be higher. However, it won't be backed by actual liquidity, thus the liquidity providers that exit earlier will get a bigger share of the underlying assets. The last liquidity provider won't be able to get their entire share.

Code Snippet

See Vulnerability Detail

Tool used

Manual Review

Recommendation

In the `_payment` function, consider updating `lien.amount` after the `beforePayment` call:

```
--- a/src/LienToken.sol
+++ b/src/LienToken.sol
@@ -614,12 +614,13 @@ contract LienToken is ERC721, ILienToken, Auth,
 ↪  TransferAgent {
       type(IPublicVault).interfaceId
     );

-    lien.amount = _getOwed(lien);
-
     address payee = getPayee(lienId);
     if (isPublicVault) {
       IPublicVault(lienOwner).beforePayment(lienId, paymentAmount);
     }
+
+    lien.amount = _getOwed(lien);
+
     if (lien.amount > paymentAmount) {
       lien.amount -= paymentAmount;
       lien.last = block.timestamp.safeCastTo32();
```

In this case, lien's slope calculation won't be affected in the `beforePayment` call and the correct slope will be removed from the slope accumulator.

Issue H-2: `LIEN_TOKEN.ownerOf(i)` should be `LIEN_TOKEN.ownerOf(liensRemaining[i])`

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/259

Found by

__141345__

Summary

In `endAuction()`, the check for public vault owner is referred to the wrong lien token id. And the actual vault lien amount is not properly recorded.

Vulnerability Detail

The lien token id should be queried is `liensRemaining[i]` instead of `i`.

Impact

`YIntercept` will not be correctly recorded. The accounting for LienToken amounts will be wrong. Hence the `totalAssets` on book will be wrong, eventually the contract and users could lose fund due to the wrong accounting.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L192-L204

Tool used

Manual Review

Recommendation

Change `LIEN_TOKEN.ownerOf(i)` to `LIEN_TOKEN.ownerOf(liensRemaining[i])`.

Issue H-3: buyoutLien() will cause the vault to fail to processEpoch()

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/245

Found by

bin2chen

Summary

LienToken#buyoutLien() did not reduce vault#liensOpenForEpoch when vault#processEpoch()will check vault#liensOpenForEpoch[currentEpoch] == uint256(0) so processEpoch() will fail

Vulnerability Detail

when create LienToken , vault#liensOpenForEpoch[currentEpoch] will ++ when repay or liquidate , vault#liensOpenForEpoch[currentEpoch] will -- and LienToken#buyoutLien() will transfer from vault to to other receiver,so liensOpenForEpoch need reduce

```
function buyoutLien(ILienToken.LienActionBuyout calldata params) external {
    ....
     /**** tranfer but not liensOpenForEpoch-- *****/
     _transfer(ownerOf(lienId), address(params.receiver), lienId);
  }
```

Impact

processEpoch() maybe fail

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L121

Tool used

Manual Review

Recommendation

```
  function buyoutLien(ILienToken.LienActionBuyout calldata params) external {
....
```

```
+   //do decreaseEpochLienCount()
+   address lienOwner = ownerOf(lienId);
+    bool isPublicVault = IPublicVault(lienOwner).supportsInterface(
+      type(IPublicVault).interfaceId
+    );
+    if (isPublicVault && !AUCTION_HOUSE.auctionExists(collateralId)) {
+        IPublicVault(lienOwner).decreaseEpochLienCount(
+          IPublicVault(lienOwner).getLienEpoch(lienData[lienId].start +
↪   lienData[lienId].duration)
+        );
+    }

    lienData[lienId].last = block.timestamp.safeCastTo32();
    lienData[lienId].start = block.timestamp.safeCastTo32();
    lienData[lienId].rate = ld.rate.safeCastTo240();
    lienData[lienId].duration = ld.duration.safeCastTo32();
    _transfer(ownerOf(lienId), address(params.receiver), lienId);
  }
```


Issue H-4: `AuctionHouse.createBid()` doesn't handle incoming payments properly.

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/223

Found by

csanuragjain, hansfriese, neila

Summary

`AuctionHouse.createBid()` doesn't handle incoming payments properly.

Vulnerability Detail

`AuctionHouse.createBid()` is used to receive bids from users and it refunds the last bidder when there is a new bidder with a higher bid amount.

But it doesn't handle the incoming payment for the new bidder here.

```
if (firstBidTime == 0) {
  auctions[tokenId].firstBidTime = block.timestamp.safeCastTo64();
} else if (lastBidder != address(0)) {
  uint256 lastBidderRefund = amount - vaultPayment;
  _handleOutGoingPayment(lastBidder, lastBidderRefund);
}

_handleIncomingPayment(tokenId, vaultPayment, address(msg.sender)); //@audit
↪  vaultPayment => amount
```

It should request the whole `amount` but `amount-currentBid` now.

Impact

`AuctionHouse.createBid()` requests a smaller amount than it should from the second bidder so the auction house protocol is lack funds.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L118

Tool used

Manual Review

Recommendation

It should request the whole amount like below.

```
_handleIncomingPayment(tokenId, amount, address(msg.sender));
```

Discussion

SantiagoGregory

The initial bid is paid to the vault, and new test coverage has confirmed multiple bid payouts are correct.

Issue H-5: Canceling an auction with 0 bids will only partially pay back the outstanding debt

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/199

Found by

0xRajeev, TurnipBoy, 0x4141, Jeiwan

Summary

Canceling an auction with 0 bids only partially pays back the outstanding debt without removing outstanding liens resulting in the collateralized NFT being locked forever.

Vulnerability Detail

Given this scenario of an auction with 0 bids and, for example, a `reservePrice` of 10 ETH and `initiatorFee` set to 10 (= 10%), with the following sequence executed:

1. The auction gets canceled
2. In `AuctionHouse.cancelAuction()`, `_handleIncomingPayment()` accepts incoming transfer of 10 ETH reserve price
3. In `_handleIncomingPayment()`, the `initiatorFee` of 10% is calculated, deducted from the 10 ETH to transfer 1 ETH to the initiator
4. The remaining 9 ETH is then used to pay back the outstanding liens (open debt)

However, the outstanding debt was initially 10 ETH (= `reservePrice`). After deducting the initiator fee, only 9 ETH remains. This means that the debt is not fully paid back. But the auction will successfully be cancelled but with outstanding debt remaining. There is also no explicit removal of liens (as there is in `endAuction`).

Impact

A borrower canceling an auction with the expected reserve price will leave behind existing unpaid liens that will make the `releaseCheck` modifier applied to `releaseToAddress()` prevent the borrower from releasing their NFT collateral successfully even after paying the reserve price during auction cancellation. The borrower's collateralized NFT is locked forever despite paying the outstanding lien amount during the auction cancellation.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L210-L224
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L253-L276
3. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L202
4. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/CollateralToken.sol#L135-L142

Tool used

Manual Review

Recommendation

The auction cancellation amount required should be reserve price + liquidation fee. On payment, remaining liens should be removed.

Issue H-6: Canceling an auction will result in a loss of borrower funds towards initiator fees

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/198

Found by

0xRajeev

Summary

The initiator fee on the current bid has already been accounted during that bid but is expected to be paid again on cancellation by the borrower.

Vulnerability Detail

Consider an active auction with a reserve price of 10 ETH and a current bid of 9 ETH. If the auction gets canceled, the `transferAmount` will be 10 ETH. Once the cancellation logic adds repayment to current bidder (see other finding reported on this issue), the initiator fees for cancellation should only be calculated on the delta payment of 1 ETH because the fees for the earlier 9 ETH bid has already been paid to the initiator. However, this is not accounted and requires the borrower to pay the initiator fees on the entire reserve price leading to overpayment towards the initiator and loss of funds for the borrower.

Impact

Canceling an auction will require paying (a lot) more initiator fees than needed resulting in a loss of funds for the borrower.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L265-L268
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L220

Tool used

Manual Review

Recommendation

The cancellation amount required should be the reserve price + liquidation fee, where the fee is calculated on `(reserveprice-currentbid)` and not the reserve price.

Issue H-7: Lien count per epoch is not updated ultimately locking the collateralized NFT

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/194

Found by

0xRajeev

Summary

The lien count per epoch is not updated, causing payments to the lien and liquidation of the lien to revert.

Vulnerability Detail

The PublicVault contract keeps track of open liens per epoch to prevent starting a new epoch with open liens. The lien count for a given epoch is decreased whenever a lien is fully paid back (either through a regular payment or a liquidation payment). However, if a lien is bought out, the lien start will be set to the current `block.timestamp` and the duration to the newly provided duration.

If the borrower now wants to make a payment to this lien, the `LienToken._payment` function will evaluate the lien's current epoch and will use a different epoch as when the lien was initially created. The attempt to then call `IPublicVault(lienOwner).decreaseEpochLienCount` will fail, as the lien count for the new epoch has not been increased yet. The same will happen for liquidations.

Impact

After a lien buyout, payments to the lien and liquidating the lien will revert, which will ultimately lock the collateralized NFT.

This will certainly prevent the borrower from making payments towards that future-epoch lien in the current epoch because `decreaseEpochLienCount()` will revert. However, even after the epoch progresses to the next one via `processEpoch()`, the `liensOpenForEpoch` for the new epoch still does not account for the previously bought out lien aligned to this new epoch because `liensOpenForEpoch` is only updated in two places: 1. `_increaseOpenLiens()` which is not called by anyone 2. `_afterCommitToLien()` <- `commitToLien()` <-- <-- `commitToLiens()` which happens only for new lien commitments

This will prevent the borrower from making payments towards the previously bought lien that aligns to the current epoch, which will force a liquidation on exceeding lien duration. Depending on when the liquidation can be triggered, if this condition is satisfied `PublicVault(owner).timeToEpochEnd()<=COLLATERAL_TOKEN.auctionWindow(`

`)` then `decreaseEpochLienCount()` will revert to prevent auctioning and lock the borrower's collateral in the protocol.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L259-L262
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L634-L636
3. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L153
4. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L399

Tool used

Manual Review

Recommendation

`liensOpenForEpoch` should be incremented when a lien is bought with a duration spilling into an epoch higher than the current one.

Issue H-8: Purchaser of a lien token may not receive payments

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/193

Found by

obront, 0xRajeev, rvierdiiev

Summary

A purchaser who buys out an existing lien via `buyoutLien()` will not receive future payments made to that lien holder if the seller had changed the lien payee via `setPayee()` and if they do not change it themselves after buying.

Vulnerability Detail

`buyoutLien()` does not reset `lienData[lienId].payee` to either 0 or to the new owner. While the ownership is transferred, the payments made in `_payment()` get their payee via `getPayee()` which returns the owner only if the `lienData[lienId].payee` is set to the zero address but returns `lienData[lienId].payee` otherwise. This will still have the value set by the previous owner who will continue to receive the payments.

Impact

The purchaser who buys out an existing lien via `buyoutLien()` will not receive future payments if the previous owner had set the payee but they do not change it via `setPayee()` themselves after buying. Given that `setPayee()` is considered optional (nothing specified in spec/docs/comments) and this reset is not part of the default flow, this will lead to a loss of purchaser's anticipated payments until they realize and reset the lien payee.

Exploit Scenario: A malicious lien seller can use `setPayee()` to set the payee to their address of choice and continue to receive payments, even after selling, if the buyer does not realize that they have to change the payee address to themselves after buying.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L619-L645
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L666-L676

3. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L678-L698

Tool used

Manual Review

Recommendation

`buyoutLien()` should reset `lienData[lienId].payee` to either the zero address or to the new owner.

Issue H-9: A malicious lien owner can exploit a reentrancy to steal LP funds

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/187

Found by

0xRajeev

Summary

A malicious lien owner can exploit a reentrancy in auctions to have their liens removed without making their payments and thus stealing LP funds.

Vulnerability Detail

A malicious lien owner can exploit a reentrancy from the callback to `decreaseYIntercept()` in `endAuction()` to call `cancelAuction()` and then take a new loan whose lien token is removed when control returns back to `endAuction()`.

Impact

PoC: https://gist.github.com/lucyoa/901a7713fded73293b5e4f9452344c5a

Exploit scenario sequence:

1. Strategist creates the public vault
2. Liquidity provider puts 50 ETH into vault
3. Malicious borrower takes a loan of 10 ETH by depositing NFT
4. Borrower buys out their own lien via `_buyoutLien` by paying 10ETH + fee
5. Borrower does not repay within lien duration to trigger liquidation
6. Borrower (or someone else) triggers `endAuction()`
7. Execution flow gets hijacked by borrower (lien token owner) inside call to `decreaseYIntercept()`:
    1. Borrower cancels auction by paying `reservePrice+initiatorFee` (this would also go to borrower if `setPayee` is set before the auction as the owner of that Lien token)
    2. Borrower releases NFT to themselves
    3. Borrower now takes another loan of 50 ETH with the same NFT, vault and commitment

8. Once control returns to `AuctionHouse.endAuction`, borrower's new loan's Lien token is deleted
9. `CollateralToken.endAuction` releases borrower's NFT to borrower
10. Malicious borrower steals 50 ETH from vault without any outstanding liens resulting in LP fund loss

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L199
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L210-L224
3. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L202

Tool used

Manual Review

Recommendation

1. Maintain a mapping of active public vaults.
2. Account for malicious lien token owners via lien buyouts.
3. Use reentrancy guards.

Issue H-10: Auctions with remaining liens will always revert causing loss of funds for the highest bidder and stuck collateral

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/185

Found by

0xRajeev

Summary

Auctions with remaining liens can never be ended because they will revert.

Vulnerability Detail

Function `endAuction` will always revert with `NOT_MINTED` error if there are any `liensRemaining` because the value passed to `LIEN_TOKEN.ownerOf` is `i` in the for loop instead of `liensRemaining[i]`.

Impact

Auctions with remaining liens can never be ended leading to loss of funds for the highest bidder and stuck collateral.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L194-L201

Tool used

Manual Review

Recommendation

Pass `liensRemaining[i]` to `LIEN_TOKEN.ownerOf`.

Issue H-11: A malicious lien buyer can DoS to cause fund loss/lock

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/181

Found by

0xRajeev

Summary

A malicious lien buyer can DoS to cause fund loss/lock of other lien holders & borrower collateral.

Vulnerability Detail

Anyone can call `buyoutLien` to purchase a lien token delivered to an arbitrary receiver. A malicious entity (even the borrower themselves) can purchase a small lien amount with its token delivered to their contract, which can implement `supportsInterface()` with arbitrary code, e.g. revert, that gets executed by the protocol in multiple places giving the attacker control at those points.

Impact

1. An attacker can revert `endAuction()` to prevent the release of collateral to the winner.
2. An attacker can revert `liquidate()` to prevent the liquidation from even starting.
3. An attacker can revert `_payment()` to prevent any lien payments from succeeding in calls to `makePayment(uint256collateralId,uint256paymentAmount)`

Thus, a malicious lien buyer can DoS to cause fund loss/lock of other lien holders & loss/lock of borrower collateral.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L195
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L381
3. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L613

Tool used

Manual Review

Recommendation

1. Maintain a mapping of active public vaults.
2. Account for malicious lien token owners via lien buyouts.
3. Use reentrancy guards.

Issue H-12: Lien buyout with new terms does not update the slope of public vaults

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/180

Found by

0xRajeev

Summary

Lien buyout with new terms does not update the slope of public vaults leading to reverts and potential insolvency of vaults.

Vulnerability Detail

In the `VaultImplementation.buyoutLien` function, a lien is bought out with new terms. Terms of a lien (last, start, rate, duration) will have changed after a buyout. This changes the slope calculation, however, after the buyout, the slope for the public vault is not updated.

Impact

There are multiple parts of the code affected by this.

1) When liquidating, the vault slope is adjusted by the liens slope (see https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L385-L387). In case of a single lien, the PublicVault.updateVaultAfterLiquidation function can revert if the lien terms have changed previously due to a buyout (https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L532). Hence, liquidations with a public vault involved, will revert.
2) The PublicVault contract calculates the implied value of a vault (ie. totalAssets) with the use of the slope value (see https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L412). As the slope value can be outdated, this leads to undervalue or overvalue of a public vault and thus vault share calculations will be incorrect.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/VaultImplementation.sol#L280-L304
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L150-L153

Tool used

Manual Review

Recommendation

Calculate the slope of the lien before the buyout in the `VautImplementation.buyoutLien` function, subtract the calculated slope value from `PublicVault.slope`, update lien terms, recalculate the slope and add the new updated slope value to `PublicVault.slope`.

Issue H-13: Auction bid that partially pays back an expired lien will revert

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/179

Found by

0xRajeev, Prefix

Summary

Auction bids will revert for a bid amount that does not fully pay back an expired lien.

Vulnerability Detail

The value of `lien.last` is updated to `block.timestamp` in several parts of the code, even for expired liens. Payments made as part of liquidations will set `lien.last` to `block.timestamp` > `lien.start+lien.duration` causing a revert in the flow shown below at step 3 when it is subtracted from `end`:

1. `IPublicVault(lienOwner).afterPayment(lienId);`
2. `slope+=LIEN_TOKEN().calculateSlope(lienId);`
3. `return(owedAtEnd-lien.amount).mulDivDown(1,end-lien.last);`

Impact

Auction bids will revert for a bid amount that does not fully pay back an expired lien. If a lien is only partially paid back, it will reach the `if` branch in `LienToken._payment`, which sets `lien.last` to the current timestamp (which itself is `>lien.start+lien.duration`, hence the liquidation) and then revert.

This affects the economic efficiency of auctions because it DoS's partial bids and therefore results in loss of funds to LPs.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L444
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L625

Tool used

Manual Review

Recommendation

Revisit the logic that updates `lien.last` in the protocol to ensure no reverts in expected flows.

Issue H-14: Epochs can be progressed during ongoing auctions to cause LP fund loss and collateral lockup

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/175

Found by

0xRajeev

Summary

The `currentEpoch` can be progressed while having an ongoing auction which will completely mess up the liquidation logic to potentially cause LP fund loss and collateral lockup.

Vulnerability Detail

The `LiquidationAccountant.claim()` function is only callable if `finalAuctionEnd` is set to 0 or the `block.timestamp` is greater than `finalAuctionEnd` (i.e. auction has ended). Furthermore, `PublicVault.processEpoch` should only be callable if there is *no* ongoing auction if a liquidation accountant is deployed in the current epoch.

`finalAuctionEnd` is set within the `LiquidationAccountant.handleNewLiquidation` function, which is called from the `AstariaRouter.liquidate` function. However, instead of providing a timestamp of the auction end, a value of `2days+1days` is given because `COLLATERAL_TOKEN.auctionWindow()` returns `2days`.

This does not achieve the intended constraint on checking for the end of the auction because it uses a fixed duration instead of a timestamp.

Impact

Epochs can be progressed during ongoing auctions to completely mess up the liquidation logic to potentially cause LP fund loss and collateral lockup.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LiquidationAccountant.sol#L67
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L244-L248
3. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L409

Tool used

Manual Review

Recommendation

Revisit the logic to use an appropriate timestamp instead of a fixed duration.

Issue H-15: Public vault depositors will receive fewer vault shares until the first payment

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/173

Found by

0xRajeev

Summary

Public vault total asset calculation is incorrect until the first payment, leading to depositors receiving fewer vault shares than expected.

Vulnerability Detail

As long as `PublicVault.last` is set to 0, the `PublicVault.totalAssets` function returns the actual ERC-20 token balance (WETH) of the public vault. Due to borrowing, this balance is reduced by the borrowed amount. Therefore, as there is no payment, this leads to depositors receiving fewer vault shares than expected.

Impact

PoC: https://gist.github.com/berndartmueller/8a71ff76c7eb8207e1f01a154a873b2c

Public vault depositors will receive fewer vault shares until the first payment.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L407

Tool used

Manual Review

Recommendation

Revisit the logic behind updating `last` when `yIntercept` and/or `slope` are updated.

Issue H-16: Payments and liquidations of multiple liens will revert and can be exploited, causing payer fund loss

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/172

Found by

0xRajeev

Summary

The `liens` array and its indexes are changed whenever a lien is fully paid back. Therefore, referencing liens by their position in the array is broken because indices change. Payments and liquidations of multiple liens within a loop will revert and can be exploited by sandwiching liens to force a payment to the attacker's lien.

Vulnerability Detail

Whenever a lien position is fully paid back, the lien is deleted from the `liens` array. This means that the array is shortened and the indices of the remaining liens are changed. This is problematic when paying back liens in a loop because the indices of the remaining liens are changed which will cause revert with an index out-of-bounds error.

One can be exploited to pay for a different (attacker's) lien position because the lien array gets shortened when another payment (by the attacker or someone else) beforehand causes a deleted lien position.

Impact

Liquidations of multiple liens and payments against a collateral token and all its liens will revert. This will result in the collateral NFT being locked forever.

Exploit scenario: A malicious junior debt holder can buy out a senior debt (sandwich the original borrower effectively) to trigger this sequence of actions. For e.g., if the `liens` array is [1, 5, 3] where the malicious junior lien holder of 3 also buys out 1 and front-runs the borrower's intended payment for 5 by repaying 1 and forcing the array to shorten to [5, 3] will make the borrower pay off their lien of 3 instead of borrower's lien of 5. Effectively, a lien holder can "force" a payment to their own lien instead of another one.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L663

2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L418
3. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L296

Tool used

Manual Review

Recommendation

Revisit and fix the entire logic that manages positions with the `liens` array addition, traversal and deletion.

Issue H-17: Incorrect operator in `AstariaRouter.isValidRefinance` can lead to borrower loss and potential liquidation

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/168

Found by

0xRajeev

Summary

Incorrect use of operator >= instead of <= in `isValidRefinance()` can lead to borrower loss and potential liquidation.

Vulnerability Detail

The check in `isValidRefinance()` for `newLien.rate>=minNewRate` is incorrect because it allows the new lien's rate of interest to be greater than maximum new rate instead of it being lesser than that value. The calculation of `uint256(lien.rate)-minInterestBPS;` actually gives `maxNewRate` and not `minNewRate`.

Impact

Refinancing is a crucial feature of the protocol to allow a borrower to refinance their loan if a certain minimum improvement of interest rate or duration is offered. This logical error allows a borrower to accidentally refinance or a malicious strategist to intentionally refinance an existing loan to a new one with a worse interest rate, leading to material loss for the borrower and potential liquidation in future.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L482-L491
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L106
3. https://docs.astaria.xyz/docs/protocol-mechanics/refinancing

Tool used

Manual Review

Recommendation

Change >= to <= in `newLien.rate>=minNewRate` in the return statement on L488.

Issue H-18: Triggering liquidations ahead of expected time leads to loss and lock of funds

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/167

Found by

0xRajeev

Summary

Triggering liquidations for a collateral after one of its liens has expired but before the auction window (default 2 days) at epoch end leads to loss and lock of funds.

Vulnerability Detail

Liquidations are allowed to be triggered for a collateral if any of its liens have exceeded their loan duration with outstanding payments. However, the liquidation logic does not execute the decrease of lien count and setting up of liquidation accountant if the time to epoch end is greater than the auction window (default 2 days). The auction proceeds nevertheless.

Impact

If liquidations are triggered before the auction window at epoch end, they will proceed to auctions without decreasing epoch lien count, without setting up the liquidation accountant for the lien and other related logic. This will, at a minimum, lead to auction proceeds going to lien owners directly instead of via the liquidation accountants (loss of funds) and the epoch unable to proceed to the next on (lock of funds and protocol halt).

Code Snippet

1.https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L388-L415

Tool used

Manual Review

Recommendation

Revisit the liquidation logic and its triggering related to the auction window and epoch end.

Issue H-19: Lack of access control in PublicVault.sol#transferWith let user call transferWithdrawReserve() multiple times to modify withdrawReserve

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/163

Found by

ctf_sec

Summary

Lack of access control in PublicVault.sol#transferWithdrawReserve let user call transferWithdrawReserve() multiple times to modify withdrawReserve

Vulnerability Detail

The function PublicVault.sol#transferWithdrawReserve() is meants to transfers funds from the PublicVault to the WithdrawProxy.

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L341-L363

However, this function has no access control, anyone can call it multiple times to modify the withdrawReserve value

Impact

A malicious actor can keep calling the function transferWthdrawReserve() before the withdrawal proxy is created.

If the underlying proxy has address(0), the transfer is not performed, but the state withdrawReserves is decremented.

Then user can invoke this function to always decrement the withdrawReserve to 0

```
// prevent transfer of more assets then are available
if (withdrawReserve <= withdraw) {
  withdraw = withdrawReserve;
  withdrawReserve = 0;
} else {
  withdrawReserve -= withdraw;
}
```

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L341-L363

Tool used

Manual Review

Recommendation

We recommend the project add requiestAuth modifier to the function

```
function transferWithdrawReserve() public {
```

We can also change the implementation by implmenting: if the underlying withdrawProxy is address(0), revert transfer.

Issue H-20: AuctionHouse#createBid bid refund logic is wrong when new bid overwrites a existing bid.

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/114

Found by

ctf_sec

Summary

AuctionHouse#createBid bid refund logic is wrong when a new bid overwrites an existing bid.

The new bidder is paying the refund for the old bidder, instead of the current auction contract refund the old bid.

Vulnerability Detail

The function AuctionHouse.sol#createBid is implemented below:

./lib/astaria-gpl/src/AuctionHouse.sol

https://github.com/AstariaXYZ/astaria-gpl/blob/0e6764f626b704fea23b5ba4c46afd963de298ca/src/AuctionHouse.sol#L94

```
function createBid(uint256 tokenId, uint256 amount) external override {
  address lastBidder = auctions[tokenId].bidder;
  uint256 currentBid = auctions[tokenId].currentBid;
  uint256 duration = auctions[tokenId].duration;
  uint64 firstBidTime = auctions[tokenId].firstBidTime;
  require(
    firstBidTime == 0 || block.timestamp < firstBidTime + duration,
    "Auction expired"
  );
  require(
    amount > currentBid + ((currentBid * minBidIncrementPercentage) / 100),
    "Must send more than last bid by minBidIncrementPercentage amount"
  );

  // If this is the first valid bid, we should set the starting time now.
  // If it's not, then we should refund the last bidder
  uint256 vaultPayment = (amount - currentBid);

  if (firstBidTime == 0) {
    auctions[tokenId].firstBidTime = block.timestamp.safeCastTo64();
  } else if (lastBidder != address(0)) {
```

```
    uint256 lastBidderRefund = amount - vaultPayment;
    _handleOutGoingPayment(lastBidder, lastBidderRefund);
  }

  _handleIncomingPayment(tokenId, vaultPayment, address(msg.sender));

  auctions[tokenId].currentBid = amount;
  auctions[tokenId].bidder = address(msg.sender);

  bool extended = false;
  // at this point we know that the timestamp is less than start + duration
↪  (since the auction would be over, otherwise)
  // we want to know by how much the timestamp is less than start + duration
  // if the difference is less than the timeBuffer, increase the duration by the
↪  timeBuffer
  if (firstBidTime + duration - block.timestamp < timeBuffer) {
    // Playing code golf for gas optimization:
    // uint256 expectedEnd =
↪  auctions[auctionId].firstBidTime.add(auctions[auctionId].duration);
    // uint256 timeRemaining = expectedEnd.sub(block.timestamp);
    // uint256 timeToAdd = timeBuffer.sub(timeRemaining);
    // uint256 newDuration = auctions[auctionId].duration.add(timeToAdd);

    //TODO: add the cap to the duration, do not let it extend beyond 24 hours
↪  extra from max duration
    uint64 newDuration = uint256(
      duration + (block.timestamp + timeBuffer - firstBidTime)
    ).safeCastTo64();
    if (newDuration <= auctions[tokenId].maxDuration) {
      auctions[tokenId].duration = newDuration;
    } else {
      auctions[tokenId].duration =
        auctions[tokenId].maxDuration -
        firstBidTime;
    }
    extended = true;
  }

  emit AuctionBid(
    tokenId,
    msg.sender,
    amount,
    lastBidder == address(0), // firstBid boolean
    extended
  );

  if (extended) {
    emit AuctionDurationExtended(tokenId, auctions[tokenId].duration);
```


```
  }
}
```

Our focus is in the codeblock below

https://github.com/AstariaXYZ/astaria-gpl/blob/0e6764f626b704fea23b5ba4c46afd963de298ca/src/AuctionHouse.sol#L112

```
if (firstBidTime == 0) {
  auctions[tokenId].firstBidTime = block.timestamp.safeCastTo64();
} else if (lastBidder != address(0)) {
  uint256 lastBidderRefund = amount - vaultPayment;
  _handleOutGoingPayment(lastBidder, lastBidderRefund);
}
```

What the code doing is that:

if the firstBidTime is 0, we set the current firsBidTime, otherwise, we refund the fund to the old bidder.

however, the crucial function _handleOutGoingPayment(lastBidder, lastBidderRefund) is implemented below

https://github.com/AstariaXYZ/astaria-gpl/blob/0e6764f626b704fea23b5ba4c46afd963de298ca/src/AuctionHouse.sol#L307

```
function _handleOutGoingPayment(address to, uint256 amount) internal {
  TRANSFER_PROXY.tokenTransferFrom(weth, address(msg.sender), to, amount);
}
```

and the implement for TRANSFER_PROXY#tokenTransferFrom is

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

then the _handleOutGoingPayment transfer "amount" of WETH from msg.sender to the address(to).

now we revisit the old bid refunding block:

```
if (firstBidTime == 0) {
  auctions[tokenId].firstBidTime = block.timestamp.safeCastTo64();
```

```
} else if (lastBidder != address(0)) {
  uint256 lastBidderRefund = amount - vaultPayment;
  _handleOutGoingPayment(lastBidder, lastBidderRefund);
}
```

if the lastBidder exits, we are letting the current new bidder who is willing to pay the higher price and wants to overwrite the current bid paying the refund for the old bidder because of this line of code:

```
TRANSFER_PROXY.tokenTransferFrom(weth, address(msg.sender), to, amount);
```

msg.sender is the current bidder, address(to) is the old bidder.

I believe this logic is not correctly implemented.

The new bidder should not refund the fund to the old bidder, whoever receives the old bidder's fund should refund the old bidder.

In this case, the old bidder's fund is used to either pay for LIEN_TOKEN or transferred to COLLATERAL_TOKEN.ownerOf(tokenId) in _handleIncomingPayment

https://github.com/AstariaXYZ/astaria-gpl/blob/0e6764f626b704fea23b5ba4c46afd963de298ca/src/AuctionHouse.sol#L254

Given the code bock

```
if (liens.length > 0) {
  for (uint256 i = 0; i < liens.length; ++i) {
    uint256 payment;
    uint256 lienId = liens[i];

    ILienToken.Lien memory lien = LIEN_TOKEN.getLien(lienId);

    if (transferAmount >= lien.amount) {
      payment = lien.amount;
      transferAmount -= payment;
    } else {
      payment = transferAmount;
      transferAmount = 0;
    }

    if (payment > 0) {
      LIEN_TOKEN.makePayment(tokenId, payment, lien.position, payer);
    }
  }
} else {
  TRANSFER_PROXY.tokenTransferFrom(
    weth,
    payer,
```

```
    COLLATERAL_TOKEN.ownerOf(tokenId),
    transferAmount
  );
}
```

Impact

The new bidder is wrongly paying the refund for the old bidder and lose money unexpectedly.

Code Snippet

https://github.com/AstariaXYZ/astaria-gpl/blob/0e6764f626b704fea23b5ba4c46afd963de298ca/src/AuctionHouse.sol#L307

https://github.com/AstariaXYZ/astaria-gpl/blob/0e6764f626b704fea23b5ba4c46afd963de298ca/src/AuctionHouse.sol#L254

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/TransferProxy.sol#L25-L32

Tool used

Manual Review

Recommendation

We recommend the project refund the old bidder's fund from LIEN_TOKEN debt position or from the COLLATERAL_TOKEN.ownerOf(tokenId) address instead of letting the new bidder paying the refund.

Issue H-21: `FlashAction` can be used to re-take a loan against the same collateral

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/109

Found by

joestakey

Summary

Users can use `FlashAction` to take a loan using that same collateral NFT.

Vulnerability Detail

`FlashAction` allows the user to unlock their underlying collateral and perform any action with the NFT as long as it is returned within the same block.

The issue is that they can use that NFT to take a new loan:

If the attacker contract implements`receiver.onFlashAction` so that it calls `AstariaRouter.commitToLiens`, this will result in srtarting a new loan, and transferring the NFT to the `COLLATERAL_TOKEN` contract. At the end of the call, the check that the NFT is returned will hence pass.

Impact

This new loan uses the same collateral as the previous one, meaning the user effectively took this new loan for free

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/CollateralToken.sol#L193-L196

Tool used

Manual Review

Recommendation

Consider checking the underlying balance of the vault in `flashAction()`, ensuring it has not changed at the end of the call.

Issue H-22: Bidder can cheat auction by placing bid much higher than reserve price when there are still open liens against a token

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/107

Found by

TurnipBoy

Summary

When a token still has open liens against it only the value of the liens will be paid by the bidder but their current bid will be set to the full value of the bid. This can be abused in one of two ways. The bidder could place a massive bid like 500 ETH that will never be outbid or they could place a bid they know will outbid and profit the difference when they're sent a refund.

Vulnerability Detail

```
uint256[] memory liens = LIEN_TOKEN.getLiens(tokenId);
uint256 totalLienAmount = 0;
if (liens.length > 0) {
  for (uint256 i = 0; i < liens.length; ++i) {
    uint256 payment;
    uint256 lienId = liens[i];

    ILienToken.Lien memory lien = LIEN_TOKEN.getLien(lienId);

    if (transferAmount >= lien.amount) {
      payment = lien.amount;
      transferAmount -= payment;
    } else {
      payment = transferAmount;
      transferAmount = 0;
    }
    if (payment > 0) {
      LIEN_TOKEN.makePayment(tokenId, payment, lien.position, payer);
    }
  }
} else {
  //@audit-issue logic skipped if liens.length > 0
  TRANSFER_PROXY.tokenTransferFrom(
    weth,
    payer,
```

```
    COLLATERAL_TOKEN.ownerOf(tokenId),
    transferAmount
  );
}
```

We can examine the payment logic inside `_handleIncomingPayment` and see that if there are still open liens against then only the amount of WETH to pay back the liens will be taken from the payer, since the else portion of the logic will be skipped.

```
uint256 vaultPayment = (amount - currentBid);

if (firstBidTime == 0) {
  auctions[tokenId].firstBidTime = block.timestamp.safeCastTo64();
} else if (lastBidder != address(0)) {
  uint256 lastBidderRefund = amount - vaultPayment;
  _handleOutGoingPayment(lastBidder, lastBidderRefund);
}
_handleIncomingPayment(tokenId, vaultPayment, address(msg.sender));

auctions[tokenId].currentBid = amount;
auctions[tokenId].bidder = address(msg.sender);
```

In `createBid`, `auctions[tokenId].currentBid` is set to `amount` after the last bidder is refunded and the excess is paid against liens. We can walk through an example to illustrate this:

Assume a token with a single lien of amount 10 WETH and an auction is opened for that token. Now a user places a bid for 20 WETH. They are the first bidder so `lastBidder=address(0)` and `currentBid=0`. `_handleIncomingPayment` will be called with a value of 20 WETH since there is no lastBidder to refund. Inside `_handleIncomingPayment` the lien information is read showing 1 lien against the token. Since `transferAmount>=lien.amount`, `payment=lien.amount`. A payment will be made by the bidder against the lien for 10 WETH. After the payment `_handleIncomingPayment` will return only having taken 10 WETH from the bidder. In the next line currentBid is set to 20 WETH but the bidder has only paid 10 WETH. Now if they are outbid, the new bidder will have to refund then 20 WETH even though they initially only paid 10 WETH.

Impact

Bidder can steal funds due to `_handleIncomingPayment` not taking enough WETH

Code Snippet

https://github.com/AstariaXYZ/astaria-gpl/blob/64acee1122a71b23eef037f69cef4c0c087241be/src/AuctionHouse.sol#L250-L304

Tool used

Manual Review

Recommendation

In `_handleIncomingPayment`, all residual transfer amount should be sent to `COLLATERAL_TOKEN.ownerOf(tokenId)`.

Issue H-23: AuctionHouse._handleIncomingPayment do not send interest to lien

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/74

Found by

rvierdiiev

Summary

AuctionHouse._handleIncomingPayment do not send interest to lien

Vulnerability Detail

When someone creates bid on auction then `AuctionHouse._handleIncomingPayment` function is called. It then send payments to lien's vaults one by one.

https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L281-L298

```
for (uint256 i = 0; i < liens.length; ++i) {
  uint256 payment;
  uint256 lienId = liens[i];

  ILienToken.Lien memory lien = LIEN_TOKEN.getLien(lienId);

  if (transferAmount >= lien.amount) {
    payment = lien.amount;
    transferAmount -= payment;
  } else {
    payment = transferAmount;
    transferAmount = 0;
  }

  if (payment > 0) {
    LIEN_TOKEN.makePayment(tokenId, payment, lien.position, payer);
  }
}
```

The problem is that it uses `lien.amount` as value that should be sent as payment to LienToken. But it also should send interests, that were accrued for a lien.

Impact

LienToken owner is underpaid with interests.

Code Snippet

Provided above.

Tool used

Manual Review

Recommendation

Consider `lien.amount+LIEN_TOKEN.getInterest(lienId)` as full amount for LienToken payment.

Issue H-24: nlrType type is not signed by strategist, which could allow fraudulent behavior as new types are added

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/72

Found by

obront

Summary

The strategist signs the merkle root, their nonce, and the deadline of all strategies to ensure that new borrowers meet their criteria. However, the lien type (`nlrType`) is not signed. Currently, the structs for the different types are unique, so there is no ability to borrow one type as another, but if struct schemas of different types overlap in the future, this will open the door for exploits.

Vulnerability Detail

When a new lien is requested, the borrower submits a Lien Request, which is filled with the parameters at which they would like to borrow. This is kept honest and aligned with the lenders intent because the merkle root, strategist nonce, and deadline are all signed by the strategist.

Because the merkle root is signed, the borrower must submit lien parameters (`nlrDetails`) that align with one of the strategies that the strategist has chosen to allow (represented as leaves in the merkle tree). The schemas of these signed structs differ depending on the validator being used, which is defined in the `nlrType` parameter.

Currently, each of the validators has a unique schema for their struct. However, if there is an overlap in the schema of the Details struct of multiple validators, where the different parameters represent different values, it opens the door to having a fraudulent lien accepted.

Here's an example of how this might work:

- Type A has a Details struct with the shape { uint8 version, bool requirementX, IAstariaRouter.LienDetails lien }.
- The lender includes in their merkle tree the following { version: 1, requirementX: true, lien: { ... maxAmount: 1 ether ... }
- Type B has a Details struct with the shape { uint8 version, bool requirementY, IAstariaRouter.LienDetails lien }
- The lender includes in their merkle tree the following strategy: { version: 1, requirementY: true, lien { ... maxAmount: 1 ether ... }

- The lender signs a merkle root including both of these strategies
- A borrower who meets requirementX but not requirementY could submit `lienDetails` with `nlrType=TypeA` and send the validation to the wrong strategy validator, thus bypassing the expected checks

Impact

As more strategy types are added, conflicts in struct schemas could open the door to fraudulent behavior.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L222-L232

Current Details struct schemas:

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/strategies/CollectionValidator.sol#L19-L24

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/strategies/UNI_V3Validator.sol#L21-L31

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/strategies/UniqueValidator.sol#L19-L25

Tool used

Manual Review

Recommendation

Include the `nlrType` in the data signed by the strategist. The easiest way to do this would be to pack it in with each leaf when assembling the leaves that will create the merkle tree:

```
function assembleLeaf(ICollectionValidator.Details memory details, address
↪  nlrType)
  public
  pure
  returns (bytes memory)
{
  return abi.encode(details, nlrType);
}

function validateAndParse... {
  ...
```

```
  leaf = keccak256(assembleLeaf(cd, params.nlrType));
}
```

Issue H-25: LienToken.createLien doesn't check if user should be liquidated and provides new loan

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/66

Found by

rvierdiiev

Summary

`LienToken.createLien` doesn't check if user should be liquidated and provides new loan if auction do not exist for collateral.

Vulnerability Detail

`LienToken.createLien` relies on `AuctionHouse` to check if new loan can be added to borrower. It assumes that if auction doesn't exist then user is safe to take new loan.

The problem is that to start auction with token that didn't pay the debt someone should call `AstariaRouter.liquidate` function. If no one did it then auction for the NFT will not exists, and `LienToken.createLien` will create new Lien to user, while he already didn't pay debt and should be liquidated.

Impact

New loan will be paid to user that didn't repay previous lien.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

Check if user can be liquidated through all of his liens positions. If not then only proceed with new loan.

Issue H-26: `PublicVault::strategistUnclaimedShares` can be manipulated

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/63

Found by

8olidity

Summary

`PublicVault::strategistUnclaimedShares` can be manipulated

Vulnerability Detail

This is a very common problem `StrategistUnclaimedShares` value is by `convertToShares(fee)`

```
function _handleStrategistInterestReward(uint256 lienId, uint256 amount)
  internal
  virtual
  override
{
  if (VAULT_FEE() != uint256(0)) {
    uint256 interestOwing = LIEN_TOKEN().getInterest(lienId);
    uint256 x = (amount > interestOwing) ? interestOwing : amount;
    uint256 fee = x.mulDivDown(VAULT_FEE(), 1000); //VAULT_FEE is a basis point
    strategistUnclaimedShares += convertToShares(fee);
  }
}
```

Continue to see `convertToShares`

```
function convertToShares(uint256 assets)
  public
  view
  virtual
  returns (uint256)
{
  uint256 supply = totalSupply(); // Saves an extra SLOAD if totalSupply is
  ↪  non-zero.

  return supply == 0 ? assets : assets.mulDivDown(supply, totalAssets());
}
```

```
function totalAssets() public view virtual override returns (uint256) {
  if (last == 0 || yIntercept == 0) {
    return ERC20(underlying()).balanceOf(address(this));
  }
  uint256 delta_t = block.timestamp - last;

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

The calculation is based on supply and totalassets.

poc

```
1. Create a `publicvault` contract
2. The attacker takes tokens from the open market and sends them to a
 ↪  `publicVault` contract
3. Since `totalassets()` is calculated based on `balanceof(address(this))`, the
 ↪  value of `totalassets` and `totalsupply` is affected if an attacker sends
 ↪  tokens.
4. An attacker can make` convertToShares()` return 0
5. `StrategistUnclaimedShares` not increase
```

Impact

`PublicVault::strategistUnclaimedShares` can be manipulated

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L513-L524 https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/ERC4626-Cloned.sol#L392-L401 https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L406-L413

Tool used

Manual Review

Recommendation

Avoid using the balance of the contract itself

Issue H-27: Auctions can end in epoch after intended, underpaying withdrawers

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/51

Found by

obront

Summary

When liens are liquidated, the router checks if the auction will complete in a future epoch and, if it does, sets up a liquidation accountant and other logistics to account for it. However, the check for auction completion does not take into account extended auctions, which can therefore end in an unexpected epoch and cause accounting issues, losing user funds.

Vulnerability Detail

The liquidate() function performs the following check to determine if it should set up the liquidation to be paid out in a future epoch:

```
if (PublicVault(owner).timeToEpochEnd() <= COLLATERAL_TOKEN.auctionWindow())
```

This function assumes that the auction will only end in a future epoch if the `auctionWindow` (typically set to 2 days) pushes us into the next epoch.

However, auctions can last up to an additional 1 day if bids are made within the final 15 minutes. In these cases, auctions are extended repeatedly, up to a maximum of 1 day.

```
if (firstBidTime + duration - block.timestamp < timeBuffer) {
  uint64 newDuration = uint256(
    duration + (block.timestamp + timeBuffer - firstBidTime)
  ).safeCastTo64();
  if (newDuration <= auctions[tokenId].maxDuration) {
    auctions[tokenId].duration = newDuration;
  } else {
    auctions[tokenId].duration =
      auctions[tokenId].maxDuration -
      firstBidTime;
  }
  extended = true;
}
```

The result is that there are auctions for which accounting is set up for them to end in the current epoch, but will actual end in the next epoch.

Impact

Users who withdrew their funds in the current epoch, who are entitled to a share of the auction's proceeds, will not be paid out fairly.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L388-L415

https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L127-L146

Tool used

Manual Review

Recommendation

Change the check to take the possibility of extension into account:

```
if (PublicVault(owner).timeToEpochEnd() <= COLLATERAL_TOKEN.auctionWindow() + 1
↪  days)
```

Issue H-28: Strategists are paid 10x the vault fee because of a math error

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/49

Found by

obront

Summary

Strategists set their vault fee in BPS (x / 10,000), but are paid out as x / 1,000. The result is that strategists will always earn 10x whatever vault fee they set.

Vulnerability Detail

Whenever any payment is made towards a public vault, `beforePayment()` is called, which calls `_handleStrategistInterestReward()`.

The function is intended to take the amount being paid, adjust by the vault fee to get the fee amount, and convert that amount of value into shares, which are added to `strategistUnclaimedShares`.

```
function _handleStrategistInterestReward(uint256 lienId, uint256 amount)
    internal
    virtual
    override
  {
    if (VAULT_FEE() != uint256(0)) {
      uint256 interestOwing = LIEN_TOKEN().getInterest(lienId);
      uint256 x = (amount > interestOwing) ? interestOwing : amount;
      uint256 fee = x.mulDivDown(VAULT_FEE(), 1000);
      strategistUnclaimedShares += convertToShares(fee);
    }
  }
```

Since the vault fee is stored in basis points, to get the vault fee, we should take the amount, multiply it by `VAULT_FEE()` and divide by 10,000. However, we accidentally divide by 1,000, which results in a 10x larger reward for the strategist than intended.

As an example, if the vault fee is intended to be 10%, we would set `VAULT_FEE=1000`. In that case, for any amount paid off, we would calculate `fee=amount*1000/1000` and the full amount would be considered a fee for the strategist.

Impact

Strategists will be paid 10x the agreed upon rate for their role, with the cost being borne by users.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L513-L524

Tool used

Manual Review

Recommendation

Change the `1000` in the `_handleStrategistInterestReward()` function to `10_000`.

Issue H-29: Claiming liquidationAccountant will reduce vault y-intercept by more than the correct amount

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/48

Found by

obront

Summary

When `claim()` is called on the Liquidation Accountant, it decreases the y-intercept based on the balance of the contract after funds have been distributed, rather than before. The result is that the y-intercept will be decreased more than it should be, siphoning funds from all users.

Vulnerability Detail

When `LiquidationAccountant.sol:claim()` is called, it uses its `withdrawRatio` to send some portion of its earnings to the `WITHDRAW_PROXY` and the rest to the vault.

After performing these transfers, it updates the vault's y-intercept, decreasing it by the gap between the expected return from the auction, and the reality of how much was sent back to the vault:

```
PublicVault(VAULT()).decreaseYIntercept(
  (expected - ERC20(underlying()).balanceOf(address(this))).mulDivDown(
    1e18 - withdrawRatio,
    1e18
  )
);
```

This rebalancing uses the balance of the `liquidationAccountant` to perform its calculation, but it is done after the balance has already been distributed, so it will always be 0.

Looking at an example:

- `expected=1ether` (meaning the y-intercept is currently based on this value)
- `withdrawRatio=0` (meaning all funds will go back to the vault)
- The auction sells for exactly 1 ether
- 1 ether is therefore sent directly to the vault
- In this case, the y-intercept should not be updated, as the outcome was equal to the expected outcome

- However, because the calculation above happens after the funds are distributed, the decrease equals `(expected-0)*1e18/1e18`, which equals `expected`

That decrease should not happen, and causing problems for the protocol's accounting. For example, when `withdraw()` is called, it uses the y-intercept in its calculation of the `totalAssets()` held by the vault, creating artificially low asset values for a given number of shares.

Impact

Every time the liquidation accountant is used, the vault's math will be thrown off and user shares will be falsely diluted.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LiquidationAccountant.sol#L62-L97

Tool used

Manual Review

Recommendation

The amount of assets sent to the vault has already been calculated, as we've already sent it. Therefore, rather than the full existing formula, we can simply call:

```
PublicVault(VAULT()).decreaseYIntercept(expected - balance)
```

Alternatively, we can move the current code above the block of code that transfers funds out (L73).

Issue H-30: liquidationAccountant can be claimed multiple times, losing a portion of all vault holders' funds

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/47

Found by

obront

Summary

The `claim()` function in LiquidationAccountant.sol does not have any protection to ensure it is only called once. Since it is a public function, any user can call it repeatedly, which will arbitrarily decrease the vault's y-intercept, causing all vault depositors to lose a portion of their funds.

Vulnerability Detail

The `claim()` function in LiquidationAccountant.sol is intended to be called after an auction is completed. The function does three things:

- Sends some portion of its balance to the `WITHDRAW_PROXY`
- Sends the rest of its balance to the vault
- Adjusts the vault's accounting, decreasing its y-intercept based on the amount of funds sent to the `WITHDRAW_PROXY` instead of the vault

Once the first two actions have been completed, there won't be any additional funds in the contract, so they won't perform any action.

However, the final action will have an impact, even if the `liquidationAccountant` has a balance of zero, because it decreases the y-intercept as follows:

```
PublicVault(VAULT()).decreaseYIntercept(
  (expected - ERC20(underlying()).balanceOf(address(this))).mulDivDown(
    1e18 - withdrawRatio,
    1e18
  )
);
```

It uses the expected amount that the auctions should have earned and subtracts the current balance of the `liquidationAccountant`, and then adjusts it by the ratio of what will be sent back to the contract.

When the balance of the contract is zero (after `claim()` has been called), it will decrease the y-intercept by `expected*(1e18-withdrawRatio)/1e18`, which will continue to reduce the y-intercept.

The arbitrary ability for any user to decrease a vault's y-intercept below the correct value risks the funds of all the holders of the vault's token.

Imagine the following scenario:

- A user calls `claim()` repeatedly until the y-intercept reaches a very low (but non-zero) number
- Any future user calls `withdraw()` on the vault with X amount of assets
- The `withdraw()` function calls `previewWithdraw()` to determine the number of shares that must be spent to withdraw the given amount of assets
- `previewWithdraw()` uses the formula `shares=assets*totalSupply()/totalAssets()`
- `totalAssets()` is calculated as `(slope*(block.timestamp)-last)+yIntercept`
- With a low y-intercept, this number will be extremely low, resulting in a high number of shares needed to retrieve a small amount of assets

Impact

A malicious (or accidental) user can cause arbitrary decreases in the y-intercept of a vault once the auction is completed, which will cause all vault depositors to lose a portion of their funds.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LiquidationAccountant.sol#L62-L97

Tool used

Manual Review

Recommendation

- Add an extra variable to LiquidationAccountant.sol with a boolean `claimed`
- Update this variable after `claim()` has been called
- Check to ensure `!claimed` at the start of `claim()` to ensure it isn't called more than once

Issue H-31: liquidationAccountant can be claimed at any time

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/46

Found by

obront

Summary

New liquidations are sent to the `liquidationAccountant` with a `finalAuctionTimestamp` value, but the actual value that is passed in is simply the duration of an auction. The `claim()` function uses this value in a require check, so this error will allow it to be called before the auction is complete.

Vulnerability Detail

When a lien is liquidated, `AstariaRouter.sol:liquidate()` is called. If the lien is set to end in a future epoch, we call `handleNewLiquidation()` on the `liquidationAccountant`.

One of the values passed in this call is the `finalAuctionTimestamp`, which updates the `finalAuctionEnd` variable in the `liquidationAccountant`. This value is then used to protect the `claim()` function from being called too early.

However, when the router calls `handleLiquidationAccountant()`, it passes the duration of an auction rather than the final timestamp:

```
LiquidationAccountant(accountant).handleNewLiquidation(
  lien.amount,
  COLLATERAL_TOKEN.auctionWindow() + 1 days
);
```

As a result, `finalAuctionEnd` will be set to 259200 (3 days).

When `claim()` is called, it requires the final auction to have ended for the function to be called:

```
require(
  block.timestamp > finalAuctionEnd || finalAuctionEnd == uint256(0),
  "final auction has not ended"
);
```

Because of the error above, `block.timestamp` will always be greater than `finalAuctionEnd`, so this will always be permitted.

Impact

Anyone can call `claim()` before an auction has ended. This can cause many problems, but the clearest is that it can ruin the protocol's accounting by decreasing the Y intercept of the vault.

For example, if `claim()` is called before the auction, the returned value will be 0, so the Y intercept will be decreased as if there was an auction that returned no funds.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L407-L410

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LiquidationAccountant.sol#L113-L120

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LiquidationAccountant.sol#L65-L69

Tool used

Manual Review

Recommendation

Adjust the call from the router to use the ending timestamp as the argument, rather than the duration:

```
LiquidationAccountant(accountant).handleNewLiquidation(
  lien.amount,
  block.timestamp + COLLATERAL_TOKEN.auctionWindow() + 1 days
);
```

Issue H-32: Incorrect fees will be charged

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/36

Found by

csanuragjain

Summary

If user has provided transferAmount which is greater than all lien.amount combined then initiatorPayment will be incorrect since it is charged on full amount when only partial was used as shown in poc

Vulnerability Detail

1. Observe the _handleIncomingPayment function
2. Lets say transferAmount was 1000
3. initiatorPayment is calculated on this full transferAmount

```
uint256 initiatorPayment = transferAmount.mulDivDown(
      auction.initiatorFee,
      100
    );
```

4. Now all lien are iterated and lien.amount is kept on deducting from transfer-Amount until all lien are navigated

```
if (transferAmount >= lien.amount) {
          payment = lien.amount;
          transferAmount -= payment;
        } else {
          payment = transferAmount;
          transferAmount = 0;
        }

        if (payment > 0) {
          LIEN_TOKEN.makePayment(tokenId, payment, lien.position, payer);
        }
      }
```

5. Lets say after loop completes the transferAmount is still left as 100
6. This means only 400 transferAmount was used but fees was deducted on full amount 500

Impact

Excess initiator fees will be deducted which was not required

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L276

Tool used

Manual Review

Recommendation

Calculate the exact amount of transfer amount required for the transaction and calculate the initiator fee based on this amount

Issue H-33: isValidRefinance checks both conditions instead of one, leading to rejection of valid refinances

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/22

Found by

obront

Summary

`isValidRefinance()` is intended to check whether either (a) the loan interest rate decreased sufficiently or (b) the loan duration increased sufficiently. Instead, it requires both of these to be true, leading to the rejection of valid refinances.

Vulnerability Detail

When trying to buy out a lien from `LienToken.sol:buyoutLien()`, the function calls `AstariaRouter.sol:isValidRefinance()` to check whether the refi terms are valid.

```
if (!ASTARIA_ROUTER.isValidRefinance(lienData[lienId], ld)) {
  revert InvalidRefinance();
}
```

One of the roles of this function is to check whether the rate decreased by more than 0.5%. From the docs:

> An improvement in terms is considered if either of these conditions is met:
>
> - The loan interest rate decrease by more than 0.5%.
> - The loan duration increases by more than 14 days.

The currently implementation of the code requires both of these conditions to be met:

```
return (
    newLien.rate >= minNewRate &&
    ((block.timestamp + newLien.duration - lien.start - lien.duration) >=
↪   minDurationIncrease)
);
```

Impact

Valid refinances that meet one of the two criteria will be rejected.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L488-L490

Tool used

Manual Review

Recommendation

Change the AND in the return statement to an OR:

```
return (
    newLien.rate >= minNewRate ||
    ((block.timestamp + newLien.duration - lien.start - lien.duration) >=
↪   minDurationIncrease)
);
```

Discussion

SantiagoGregory

Independently fixed during our own review so there's no PR specifically for this, but this is now updated to an or.

Issue H-34: isValidRefinance will approve invalid refinances and reject valid refinances due to buggy math

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/21

Found by

obront, hansfriese

Summary

The math in `isValidRefinance()` checks whether the rate increased rather than decreased, resulting in invalid refinances being approved and valid refinances being rejected.

Vulnerability Detail

When trying to buy out a lien from `LienToken.sol:buyoutLien()`, the function calls `AstariaRouter.sol:isValidRefinance()` to check whether the refi terms are valid.

```
if (!ASTARIA_ROUTER.isValidRefinance(lienData[lienId], ld)) {
  revert InvalidRefinance();
}
```

One of the roles of this function is to check whether the rate decreased by more than 0.5%. From the docs:

> An improvement in terms is considered if either of these conditions is met:
>
> - The loan interest rate decrease by more than 0.5%.
> - The loan duration increases by more than 14 days.

The current implementation of the function does the opposite. It calculates a `minNewRate` (which should be `maxNewRate`) and then checks whether the new rate is greater than that value.

```
uint256 minNewRate = uint256(lien.rate) - minInterestBPS;
return (newLien.rate >= minNewRate ...
```

The result is that if the new rate has increased (or decreased by less than 0.5%), it will be considered valid, but if it has decreased by more than 0.5% (the ideal behavior) it will be rejected as invalid.

Impact

- Users can perform invalid refinances with the wrong parameters.

- Users who should be able to perform refinances at better rates will not be able to.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L482-L491

Tool used

Manual Review

Recommendation

Flip the logic used to check the rate to the following:

```
uint256 maxNewRate = uint256(lien.rate) - minInterestBPS;
return (newLien.rate <= maxNewRate...
```

Issue H-35: deposit() and mint() functions do not work

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/2

Found by

Bnke0x0

Summary

Vulnerability Detail

Impact

The `ERC4626-Cloned` contract inherits from the `ERC20Cloned` and `ERC4626Base` contracts. When the user calls the deposit and mint functions of the `ERC4626-Cloned` contract, the deposit and mint functions of the `ERC20Cloned` or `ERC4626Base` contracts are called.

The deposit and mint functions of the `ERC20Cloned` or `ERC4626Base` contracts will call the deposit and mint functions of the `ERC4626-Cloned` contract. The `ERC4626-Cloned` contract inherits from the ERC4626 contract, that is, the deposit and mint functions of the ERC4626 contract will be called.

The deposit and mint functions of the ERC4626 contract will call the safeTransferFrom function. Since the caller is the `ERC4626-Cloned` contract, msg.sender will be the `ERC4626-Cloned` contract. And because the user calls the deposit and mint functions of the `ERC4626-Cloned` contract without transferring tokens to the `ERC4626-Cloned` contract and approving the call will fail.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/ERC4626-Cloned.sol#L305-L322

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

    // Need to transfer before minting or ERC777s could reenter.
    ERC20(underlying()).safeTransferFrom(msg.sender, address(this), assets);
```

```
    _mint(receiver, shares);

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

    afterDeposit(assets, shares);
  }`
```

https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/ERC4626-Cloned.sol#L324-L339

```
`   function mint(uint256 shares, address receiver)
    public
   virtual
   returns (uint256 assets)
 {
   assets = previewMint(shares); // No need to check for rounding error,
    ↪  previewMint rounds up.

   // Need to transfer before minting or ERC777s could reenter.
  ERC20(underlying()).safeTransferFrom(msg.sender, address(this), assets);

   _mint(receiver, shares);

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

   afterDeposit(assets, shares);
 }`
```

Tool used

Manual Review

Recommendation

In the deposit and mint functions of the `ERC4626-Cloned` contract, add code for the user to transfer tokens and approve the use of tokens in the Astaria contract.

Issue M-1: Use safeTransferFrom() instead of transferFrom() for outgoing erc721 transfer

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/293

Found by

pashov, neila, Bnke0x0, 8olidity, Sm4rty, Rohan16, cryptphi, Nyx, seyni, yixxas, rvierdiiev, peanuts

Summary

It is recommended to use safeTransferFrom() instead of transferFrom() when transferring ERC721s out of the vault.

Vulnerability Detail

1. The transferFrom() method is used instead of safeTransferFrom(), which I assume is a gas-saving measure. I however argue that this isn't recommended because:

- OpenZeppelin's documentation discourages the use of transferFrom(); use safeTransferFrom() whenever possible
- The recipient could have logic in the onERC721Received() function, which is only triggered in the safeTransferFrom() function and not in transferFrom(). A notable example of such contracts is the Sudoswap pair:

Impact

While unlikely because the recipient is the function caller, there is the potential loss of NFTs should the recipient is unable to handle the sent ERC721s. IERC721(underlyingAsset).transferFrom(address(this), releaseTo, assetId); nft.transferFrom(address(this), address(receiver), tokenId);

Code Snippet

```
nft.transferFrom(address(this), address(receiver), tokenId);
 IERC721(underlyingAsset).transferFrom(address(this), releaseTo, assetId);
```

Tool used

Manual Review

Recommendation

Use safeTransferFrom() when sending out the NFT from the vault.

```
- IERC721(_erc721Address).transferFrom(address(this), msg.sender, _id);
+ IERC721(_erc721Address).safeTransferFrom(address(this), msg.sender, _id);
```

Note that the vault would have to inherit the IERC721Receiver contract if the change is applied to Transfer.sol as well.

Issue M-2: Auction still exists even paid off the loan

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/285

Found by

__141345__

Summary

The auction will not be cancelled even the loan is paid off. As a result, the collateral will still be liquidated. If the bid is very low, the lenders of other positions of the collateral could have fund loss.

Vulnerability Detail

Consider the following case: A collateral is taking 2 different loan, 1 for a private vault with duration of 15 days for $100, 1 for a public vault with duration of 30 days for $1,000. Say after 15 days, the loan is not paid back, it will be liquidated and put on auction.

Now, if the borrower pays back the first loan, but the auction won't be cancelled and still be alive. Then the collateral could be liquidated and the 2nd loan lender could suffer fund loss.

Impact

Some lenders could loss fund they do not deserve.

Code Snippet

The auction won't be changed even after paid off the 1st lien token loan. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L594-L649

Tool used

Manual Review

Recommendation

Cancel the auction is all the lien token loans are paid off.

Issue M-3: Re-entrancy

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/284

Found by

0x4141, HonorLt

Summary

The protocol does not have safety checks against re-entrancy attacks.

Vulnerability Detail

The protocol is quite big and complex and there are many outside interactions with users or other contracts containing callbacks. For example, users can `flashAction` their NFTs if they return them in the same transaction. I believe this makes it possible to re-use the same collateral twice. For instance, when the control is given back to the user:

```
require(
  receiver.onFlashAction(IFlashAction.Underlying(addr, tokenId), data) ==
    keccak256("FlashAction.onFlashAction"),
  "flashAction: callback failed"
);
```

a malicious actor can `commitToLien` again (combining with issues that the `ecrecover` might be bypassed and a strategist nonce is not invalidated). Because the user already owns the collateral token, the contract will skip the minting and return:

```
function onERC721Received(
  address operator_,
  address from_,
  uint256 tokenId_,
  bytes calldata data_
) external override returns (bytes4) {
  uint256 collateralId = msg.sender.computeId(tokenId_);

  (address underlyingAsset, ) = getUnderlying(collateralId);
  if (underlyingAsset == address(0)) {
    ...
  }
  return IERC721Receiver.onERC721Received.selector;
}
```

Impact

I believe it might be possible to reuse the same collateral twice. I was short on time to PoC this and there might be more places where re-entrancy can be exploited. The codebase is just quite complicated to make a sufficient mental model about the intended behavior.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/CollateralToken.sol#L149-L197

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/CollateralToken.sol#L259-L296

Tool used

Manual Review

Recommendation

Add re-entrancy guard to critical functions that contain external interactions.

Issue M-4: Liquidity providers can lose funds when a withdraw proxy is not set for an epoch

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/277

Found by

Jeiwan

Summary

Liquidity providers can lose funds when a withdraw proxy is not set for an epoch

Vulnerability Detail

The `transferWithdrawReserve` function of `PublicVault` sends withdrawal reserves to a `WithdrawProxy` (PublicVault.sol#L341) and subtracts transferred amount from the reserves. However, if a withdraw proxy is not set for an epoch, there's a false positive: `withdrawReserve` is updated but no funds are actually transferred:

```
function transferWithdrawReserve() public {
  // check the available balance to be withdrawn
  uint256 withdraw = ERC20(underlying()).balanceOf(address(this));
  emit TransferWithdraw(withdraw, withdrawReserve);

  // prevent transfer of more assets then are available
  if (withdrawReserve <= withdraw) {
    withdraw = withdrawReserve;
    withdrawReserve = 0;
  } else {
    withdrawReserve -= withdraw;
  }
  emit TransferWithdraw(withdraw, withdrawReserve);

  address currentWithdrawProxy = withdrawProxies[currentEpoch - 1]; //
  // prevents transfer to a non-existent WithdrawProxy
  // withdrawProxies are indexed by the epoch where they're deployed
  if (currentWithdrawProxy != address(0)) { // @audit false positive:
↪   transferring is skipped silently
    ERC20(underlying()).safeTransfer(currentWithdrawProxy, withdraw);
    emit WithdrawReserveTransferred(withdraw);
  }
}
```

Impact

Liquidity providers might not be able to withdraw liquidity they requested because it wasn't transferred to a WithdrawProxy due to a mistake, yet accounting was updated.

Code Snippet

See Vulnerability Detail

Tool used

Manual Review

Recommendation

Consider reverting in the case when no withdraw proxy is set for the current epoch.

Discussion

SantiagoGregory

We now check that the withdrawProxy for the current epoch (technically previous epoch) exists before transferring.

Issue M-5: Auction#reservePrice maybe less than required

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/266

Found by

bin2chen

Summary

Auction#reservePrice without the addition of the initiatorFee, maybe less than required

Vulnerability Detail

The auction will pay a #initiatorFee when paying, but the fee is not added to #reservePrice when creating auction, so when cancel, the price will not enough

Impact

can't clear all lienToken

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L276

```
function _handleIncomingPayment(
  uint256 tokenId,
  uint256 transferAmount,
  address payer
) internal {
  require(transferAmount > uint256(0), "cannot send nothing");
  Auction storage auction = auctions[tokenId];

  uint256 initiatorPayment = transferAmount.mulDivDown(
    auction.initiatorFee,
    100
  );
  TRANSFER_PROXY.tokenTransferFrom(
    weth,
    payer,
    auction.initiator,
    initiatorPayment
  );
```

```
  transferAmount -= initiatorPayment; /******** pay initiatorFee ********/
```

Tool used

Manual Review

Recommendation

```
  function createAuction(
    uint256 tokenId,
    uint256 duration,
    address initiator,
    uint256 initiatorFee
  ) external requiresAuth returns (uint256 reserve) {
    (reserve, ) = LIEN_TOKEN.stopLiens(tokenId);

    Auction storage newAuction = auctions[tokenId];
    newAuction.duration = duration.safeCastTo64();
-   newAuction.reservePrice = reserve;
+    newAuction.reservePrice = reserve + reserve.mulDivDown(
                                                    auction.initiatorFee,
                                                    100
                                                );
    newAuction.initiator = initiator;
    newAuction.initiatorFee = initiatorFee;
    newAuction.firstBidTime = block.timestamp.safeCastTo64();
    newAuction.maxDuration = (duration + 1 days).safeCastTo64();
    newAuction.currentBid = 0;

    emit AuctionCreated(tokenId, duration, reserve);
  }
```

Issue M-6: new loans "max duration" is not restricted

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/264

Found by

bin2chen

Summary

document : " Epochs PublicVaults operate around a time-based epoch system. An epoch length is defined by the strategist that deploys the PublicVault. The duration of new loans is restricted to not exceed the end of the next epoch. For example, if a PublicVault is 15 days into a 30-day epoch, new loans must not be longer than 45 days. " but more than 2 epoch's duration can be added

Vulnerability Detail

the max duration is not detected. add success when > next epoch

#AstariaTest#testBasicPublicVaultLoan

```
function testBasicPublicVaultLoan() public {

IAstariaRouter.LienDetails memory standardLien2 =
  IAstariaRouter.LienDetails({
    maxAmount: 50 ether,
    rate: (uint256(1e16) * 150) / (365 days),
    duration: 50 days,  /****** more then 14 * 2 *******/
    maxPotentialDebt: 50 ether
  });

  _commitToLien({
    vault: publicVault,
    strategist: strategistOne,
    strategistPK: strategistOnePK,
    tokenContract: tokenContract,
    tokenId: tokenId,
    lienDetails: standardLien2, /**** use standardLien2 ****/
    amount: 10 ether,
    isFirstLien: true
  });
}
```

Impact

Too long duration

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/VaultImplementation.sol#L209

Tool used

Manual Review

Recommendation

PublicVault#_afterCommitToLien

```
  function _afterCommitToLien(uint256 lienId, uint256 amount)
    internal
    virtual
    override
  {
    // increment slope for the new lien
    unchecked {
      slope += LIEN_TOKEN().calculateSlope(lienId);
    }

    ILienToken.Lien memory lien = LIEN_TOKEN().getLien(lienId);

    uint256 epoch = Math.ceilDiv(
      lien.start + lien.duration - START(),
      EPOCH_LENGTH()
    ) - 1;

+   require(epoch <= currentEpoch + 1,"epoch max <= currentEpoch + 1");

    liensOpenForEpoch[epoch]++;
    emit LienOpen(lienId, epoch);
  }
```

Issue M-7: cancelAuction() can cancel "the completed Auction"

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/239

Found by

bin2chen

Summary

AuctionHouse#cancelAuction() can cancel "the completed Auction"

Vulnerability Detail

When the auction has ended (time is up), and the bid amount is less than reservePrice, then it still can #cancel(), this does not make sense, after the end of the auction should only #endAuction()

Impact

after the end of the auction should cancel

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L210

```
function cancelAuction(uint256 auctionId, address canceledBy)
  external
  requiresAuth
{
  require(
    auctions[auctionId].currentBid < auctions[auctionId].reservePrice,
    "cancelAuction: Auction is at or above reserve"
  );
/**** only check price , no check if end ****/
```

Tool used

Manual Review

Recommendation

require not end

```
function cancelAuction(uint256 auctionId, address canceledBy)
  external
  requiresAuth
{
  require(
    auctions[auctionId].currentBid < auctions[auctionId].reservePrice,
    "cancelAuction: Auction is at or above reserve"
  );
      require(
    block.timestamp <
      auctions[auctionId].firstBidTime + auctions[auctionId].duration,
    "Auction had completed"
  );
```

Issue M-8: LienToken's `calculateSlope` might panic

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/235

Found by

sorrynotsorry

Summary

LienToken's `calculateSlope` might panic

Vulnerability Detail

LienToken's `calculateSlope` returns the computed rate for a specified lien. However, the return is not validated against the linen's being active and the `last` value.

The function is as below;

```
function calculateSlope(uint256 lienId) public view returns (uint256) {
  Lien memory lien = lienData[lienId];
  uint256 end = (lien.start + lien.duration);
  uint256 owedAtEnd = _getOwed(lien, end);
  return (owedAtEnd - lien.amount).mulDivDown(1, end - lien.last);
}
```

Permalink

If the lien is not active and the payout is executed at the `end` of the lien then `(end-lien.last)` will be 0 causing the function to panic.

Impact

Incorrect accounting

Code Snippet

```
function calculateSlope(uint256 lienId) public view returns (uint256) {
  Lien memory lien = lienData[lienId];
  uint256 end = (lien.start + lien.duration);
  uint256 owedAtEnd = _getOwed(lien, end);
  return (owedAtEnd - lien.amount).mulDivDown(1, end - lien.last);
}
```

Permalink

Tool used

Manual Review

Recommendation

Validate accordingly;

```
function calculateSlope(uint256 lienId) public view returns (uint256) {
  Lien memory lien = lienData[lienId];
  require(lien.active,"err");
  uint256 end = (lien.start + lien.duration);
  require(end != lien.last,"err");
  uint256 owedAtEnd = _getOwed(lien, end);
  return (owedAtEnd - lien.amount).mulDivDown(1, end - lien.last);
}
```

Issue M-9: _deleteLienPosition can be called by anyone to delete any lien they wish

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/233

Found by

obront, 0x0, tives, TurnipBoy, ctf_sec, zzykxx, yixxas

Summary

`_deleteLienPosition` is a public function that doesn't check the caller. This allows anyone to call it an remove whatever lien they wish from whatever collateral they wish

Vulnerability Detail

```
function _deleteLienPosition(uint256 collateralId, uint256 position) public {
  uint256[] storage stack = liens[collateralId];
  require(position < stack.length, "index out of bounds");

  emit RemoveLien(
    stack[position],
    lienData[stack[position]].collateralId,
    lienData[stack[position]].position
  );
  for (uint256 i = position; i < stack.length - 1; i++) {
    stack[i] = stack[i + 1];
  }
  stack.pop();
}
```

`_deleteLienPosition` is a `public` function and doesn't validate that it's being called by any permissioned account. The result is that anyone can call it to delete any lien that they want. It wouldn't remove the lien data but it would remove it from the array associated with `collateralId`, which would allow it to pass the `CollateralToken.sol releaseCheck` and the underlying to be withdrawn by the user.

Impact

All liens can be deleted completely rugging lenders

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L651-L664

Tool used

Manual Review

Recommendation

Change `_deleteLienPosition` to `internal` rather than `public`.

Issue M-10: `commitToLiens` always reverts

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/204

Found by

0xRajeev, bin2chen, HonorLt, Jeiwan, rvierdiiev

Summary

The function `commitToLiens()` always reverts at the call to `_returnCollateral()` which prevents borrowers from depositing collateral and requesting loans in the protocol.

Vulnerability Detail

The collateral token with `collateralId` is already minted directly to the caller (i.e. borrower) in `commitToLiens()` at the call to `_transferAndDepositAsset()` function. That's because while executing `_transferAndDepositAsset` the NFT is transferred to `COLLATERAL_TOKEN` whose `onERC721Received` mints the token with `collateralId` to borrower (`from` address) and not the `operator_` (i.e. `AstariaRouter`) because `operator_!=from_`.

However, the call to `_returnCollateral()` in `commitToLiens()` incorrectly assumes that this has been minted to the operator and attempts to transfer it to the borrower which will revert because the `collateralId` is not owned by `AstariaRouter` as it has already been transferred/minted to the borrower.

Impact

The function `commitToLiens()` always reverts, preventing borrowers from depositing collateral and requesting loans in the protocol, thereby failing to bootstrap its core NFT lending functionality.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L244-L274
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L578-L587
3. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/CollateralToken.sol#L282-L284
4. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L589-L591

Tool used

Manual Review

Recommendation

Remove the call to `_returnCollateral()` in `commitToLiens()`.

Issue M-11: Canceling an auction does not refund the current highest bidder

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/202

Found by

0xRajeev, Prefix, neila, bin2chen, csanuragjain, hansfriese, chainNue, TurnipBoy, minhquanym, Jeiwan, peanuts

Summary

If the collateral token owner cancels the active auction and repays outstanding debt, the current highest bidder will not be refunded and loses their funds.

Vulnerability Detail

The `AuctionHouse.createBid()` function refunds the previous bidder if there is one. The same logic would also be necessary in the `AuctionHouse.cancelAuction()` function but is missing.

Impact

If the collateral token owner cancels the active auction and repays outstanding debt (`reservePrice`), the current highest bidder will not be refunded and will therefore lose their funds which can also be exploited by a malicious borrower.

Potential exploit scenario: A malicious borrower can let the loan expire without repayment, trigger an auction, let bids below reserve price, and (hope to) front-run any bid >= reserve price to cancel the auction which effectively lets the highest bidder pay out (most of) the liens instead of the borrower.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L113-L116
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L210-L224

Tool used

Manual Review

Recommendation

Add the refund logic (via `_handleOutGoingPayment()` to the current bidder) in the cancel auction flow similar to the create bid auction flow.

Issue M-12: Extension logic incorrectly extends the auction by an additional amount of existing duration

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/201

Found by

0xRajeev, Prefix, bin2chen, minhquanym, yixxas

Summary

Incorrect auction extension logic extends the auction by an additional amount of the previous duration instead of extending it by 15 minutes.

Vulnerability Detail

The calculation of `newDuration` incorrectly adds `duration` in the auction extension logic. This causes the new duration to be extended by an additional amount of the existing duration, instead of an additional 15 minutes (`timeBuffer`), when a bid is created in the last 15 mins of the existing auction duration.

Impact

Delayed payout of funds/collateral upon auction completion only after the newly extended duration.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L135-L137

Tool used

Manual Review

Recommendation

Change the calculation to `uint64newDuration=uint256(block.timestamp+timeBuffer-firstBidTime).safeCastTo64();`

Issue M-13: Public vaults can become insolvent because of missing `yIntercept` update

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/197

Found by

zzykxx, 0xRajeev

Summary

The deduction of `yIntercept` during payments is missing in `beforePayment()` which can lead to vault insolvency.

Vulnerability Detail

`yIntercept` is declared as "sum of all LienToken amounts" and documented elsewhere as "yIntercept (virtual assets) of a PublicVault". It is used to calculate the total assets of a public vault as: `slope.mulDivDown(delta_t,1)+yIntercept`.

It is expected to be updated on deposits, payments, withdrawals, liquidations. However, the deduction of `yIntercept` during payments is missing in `beforePayment()`. As noted in the function's Natspec:

```
/**
  * @notice Hook to update the slope and yIntercept of the PublicVault on
↪  payment.
  * The rate for the LienToken is subtracted from the total slope of the
↪  PublicVault, and recalculated in afterPayment().
  * @param lienId The ID of the lien.
  * @param amount The amount paid off to deduct from the yIntercept of the
↪  PublicVault.
  */
```

the amount of payment should be deducted from `yIntercept` but is missing.

Impact

PoC: https://gist.github.com/berndartmueller/477cc1026d3fe3e226795a34bb8a903a

This missing update will inflate the inferred value of the public vault corresponding to its actual value leading to eventual insolvency because of resulting protocol miscalculations.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L427-L442

Tool used

Manual Review

Recommendation

Update `yIntercept` in `beforePayment()` by the `amount` value.

Discussion

androolloyd

tagging @SantiagoGregory but i believe this is a documentation error, will address

Issue M-14: `LienToken.buyoutLien` will always revert

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/196

Found by

0xRajeev, supernova, neila, yixxas, 8olidity, ctf_sec, zzykxx, cccz, rvierdiiev

Summary

`buyoutLien()` will always revert, preventing the borrower from refinancing.

Vulnerability Detail

`buyoutFeeDenominator` is `0` without a setter which will cause `getBuyoutFee()` to revert in the `buyoutLien()` flow.

Impact

Refinancing is a crucial feature of the protocol to allow a borrower to refinance their loan if a certain minimum improvement of interest rate or duration is offered. The reverting `buyoutLien()` flow will prevent the borrower from refinancing and effectively lead to loss of their funds due to lock-in into currently held loans when better terms are available.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L71
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L456
3. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L377
4. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L132

Tool used

Manual Review

Recommendation

Initialize the buyout fee numerator and denominator in `AstariaRouter` and add their setters to `file()`.

Issue M-15: `LienToken.createLien` may prevent liens that satisfy their terms of `maxPotentialDebt`

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/192

Found by

0xRajeev, hansfriese

Summary

The `potentialDebt` calculation in `createLien` is incorrect.

Vulnerability Detail

The calculated `potentialDebt` is effectively `(impliedRate+totalDebt)*params.terms.duration` because `getImpliedRate()` returns `impliedRate=impliedRate.mulDivDown(1,totalDebt);`. The calculated `potentialDebt` because of multiplying `totalDebt` by duration is significantly higher than it actually is and so will fail the `params.terms.maxPotentialDebt` check and revert.

Impact

This will cause DoS on valid lien creation.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L256-L260
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L526

Tool used

Manual Review

Recommendation

Have `getImpliedRate()` *not* do `impliedRate=impliedRate.mulDivDown(1,totalDebt);` and calculate `potentialDebt` as `totalDebt*(1+impliedRate*params.terms.duration*mulDivDown(1,INTEREST_DENOMINATOR)`.

Issue M-16: A payment made towards multiple liens causes the borrower to lose funds to the payee

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/190

Found by

obront, 0xRajeev, TurnipBoy, Jeiwan, zzykxx

Summary

A payment made towards multiple liens is entirely consumed for the first one causing the borrower to lose funds to the payee.

Vulnerability Detail

A borrower can make a bulk payment against multiple liens for a collateral hoping to pay more than one at a time using `makePayment(uint256collateralId,uint256paymentAmount)` where the underlying `_makePayment()` loops over the open liens attempting to pay off more than one depending on the `totalCapitalAvailable` provided.

However, the entire `totalCapitalAvailable` is provided via `paymentAmount` in the call to `_payment()` in the first iteration which transfers that completely to the payee in its logic even if it exceeds that `lien.amount`. That total amount is returned as `capitalSpent` which makes the `paymentAmount` for next iteration equal to `0`.

Impact

Only the first lien is paid off and the entire payment is sent to its payee. The remaining liens remain unpaid. The payment maker (i.e. borrower) loses funds to the payee.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L387-L389
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L410-L424
3. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L630-L645

Tool used

Manual Review

Recommendation

Add `paymentAmount-=lien.amount` in the `else` block of `_payment()`.

Issue M-17: Incorrect `LienToken.changeInSlope` calculation can lead to vault insolvency

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/189

Found by

0xRajeev, hansfriese

Summary

The calculation of `newSlope` in `changeInSlope()` is incorrect which can lead to vault insolvency.

Vulnerability Detail

Contrary to the `changeInSlope` function, the `calculateSlope` function uses the `_getOwed` function to calculate the owed amount (incl. the interest). The interest is calculated with the `_getInterest` function. This function takes care of the `lien.last` and also if a lien is expired already. This logic is completely missing in the `changeInSlope` for the `newSlope` calculation which makes it incorrect. Also, very importantly, in the `changeInSlope` function, the `INTEREST_DENOMINATOR` is missing which makes the value inflated causing an underflow error in the last line of `changeInSlope` function: `slope=oldSlope-newSlope;`. `oldslope`, which accounts for the `INTEREST_DENOMINATOR`, is less than `newSlope`.

Impact

The incorrect `changeInSlope()` calculation can lead to reverts and vault insolvency because users cannot determine the implicit value of vaults while interacting with it as borrowers or lenders.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L453-L469
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L440-L445

Tool used

Manual Review

Recommendation

Make `changeInSlope()` consistent with `calculateSlope()` by implementing a separate helper function to calculate the interest accounting for all the parameters and reusing it in both places.

Issue M-18: `LiquidationAccountant.claim()` can be called by anyone causing vault insolvency

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/188

Found by

0xRajeev, TurnipBoy

Summary

`LiquidationAccountant.claim()` can be called by anyone to reduce the implied value of a public vault.

Vulnerability Detail

`LiquidationAccountant.claim()` is called by the `PublicVault` as part of the `processEpoch()` flow. But it has no access control and can be called by anyone and any number of times. If called after `finalAuctionEnd`, one will be able to trigger `decreaseYIntercept()` on the vault even if they cannot affect fund transfer to withdrawing liquidity providers and the PublicVault.

Impact

This allows anyone to manipulate the `yIntercept` of a public vault by triggering the `claim()` flow after liquidations resulting in vault insolvency.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LiquidationAccountant.sol#L65-L97

Tool used

Manual Review

Recommendation

Allow only vault to call `claim()` by requiring authorizations.

Discussion

SantiagoGregory

Our updated LiquidationAccountant implementation (now moved to WithdrawProxy) tracks a hasClaimed bool to make sure claim() is only called once (we also now block claim() from being called until after finalAuctionEnd).

Issue M-19: Auctions run for less time than intended

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/186

Found by

obront, 0xRajeev, csanuragjain, hansfriese, chainNue, peanuts

Summary

Auctions run for less time than intended causing them to be not economically efficient for the lenders, thus causing a suboptimal credit of liquidation funds to them.

Vulnerability Detail

From the documentation/comments, we can infer that `firstBidTime` is supposed to be `//Thetimeofthefirstbid` and duration is supposed to be `//Thelengthoftimetoruntheauctionfor,afterthefirstbidwasmade.`

However, when an auction is created in `createAuction()`, the auction's `firstBidTime` is incorrectly initialized as `block.timestamp.safeCastTo64()` instead of `0`. This is premature initialization because the auction was only created here and no bid has been made yet via `createBid()`. The code in `createBid()` which check for `firstBidTime==0`is more evidence that the initialization in `createAuction()` is incorrect.

Impact

This causes auctions to run for less time than intended if the first bid comes at a much later time after the auction was created. A shorter auction time could potentially allow fewer bids and cause it to be not economically efficient for the lenders, thus causing a suboptimal credit of liquidation funds to them.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/interfaces/IAuctionHouse.sol#L12-L13
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L80
3. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L166-L176

Tool used

Manual Review

Recommendation

`createAuction()` should initialize `firstBidTime=0`.

Discussion

androolloyd

will fix but its a convention issue we want auctions to start immediately not on first bid

Issue M-20: `VaultImplementation._validateCommitment` may prevent liens that satisfy their terms of `maxPotentialDebt`

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/182

Found by

obront, 0xRajeev, hansfriese, tives, Jeiwan, zzykxx, rvierdiiev

Summary

The calculation of `potentialDebt` in `VaultImplementation._validateCommitment()` is incorrect and will cause a DoS to legitimate borrowers.

Vulnerability Detail

The calculation of potentialDebt in `VaultImplementation._validateCommitment()` is incorrect because it computes `uint256potentialDebt=seniorDebt*(ld.rate+1)*ld.duration;` which incorrectly adds a factor of `ld.duration` to `seniorDebt` thus making the potential debt much higher by that factor than it will be. The use of `INTEREST_DENOMINATOR` and implied lien rate is also missing here.

Impact

Liens that would have otherwise satisfied the constraint of `potentialDebt<=ld.maxPotentialDebt` will fail because of this miscalculation and will cause a DoS to legitimate borrowers and likely all of them.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/VaultImplementation.sol#L221-L225
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L256-L262

Tool used

Manual Review

Recommendation

Change the calculation to `uint256potentialDebt=seniorDebt*(ld.rate*ld.duration+1).mulDivDown(1,INTEREST_DENOMINATOR);`. This should also consider the implied rate of all the liens against the collateral instead of only this lien.

Issue M-21: Anyone can deposit and mint withdrawal proxy shares to capture distributed yield from borrower interests

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/176

Found by

0xRajeev, TurnipBoy, bin2chen

Summary

Anyone can deposit and mint Withdrawal proxy shares by directly interacting with the base `ERC4626Cloned` contract's functions, allowing them to capture distributed yield from borrower interests.

Vulnerability Detail

The `WithdrawProxy` contract extends the `ERC4626Cloned` vault contract implementation. The `ERC4626Cloned` contract has the functionality to deposit and mint vault shares. Usually, withdrawal proxy shares are only distributed via the `WithdrawProxy.mint` function, which is only called by the `PublicVault.redeemFutureEpoch`function. Anyone can deposit WETH into a deployed Withdraw proxy to receive shares, wait until assets (WETH) are deposited via the `PublicVault.transferWithdrawReserve` or `LiquidationAccountant.claim` function and then redeem their shares for WETH assets.

Impact

By depositing/minting directly to the Withdraw proxy, one can get interest yield on-demand without being an LP and having capital locked for epoch(s). This may potentially be timed in a way to deposit/mint only when we know that interest yields are being paid by a borrower who is not defaulting on their loan. The returns are diluted for the LPs at the expense of someone who directly interacts with the underlying proxy.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/ERC4626-Cloned.sol#L305-L339
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L198

Tool used

Manual Review

Recommendation

Overwrite the `ERC4626Cloned.afterDeposit` function and revert to prevent public deposits and mints.

Issue M-22: Minting public vault shares while the protocol is paused can lead to LP fund loss

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/174

Found by

0xRajeev

Summary

Assets can be deposited into public vaults by LPs with `PublicVault.mint` function to bypass a possible paused protocol.

Vulnerability Detail

The PublicVault contract prevents calls to the `PublicVault.deposit` function while the protocol is paused by using the `whenNotPaused` modifier.

The `PublicVault` contract extends the `ERC4626Cloned` contract, which has two functions to deposit assets into the vault: the `deposit` function and the `mint` function. The latter function, however, is not overwritten in the PublicVault contract and therefore lacks the appropriate `whenNotPaused` modifier.

Impact

LPs can deposit assets into public vaults with the `PublicVault.mint` function to bypass a possible paused protocol. This can lead to LP fund loss depending on the reason for the protocol pause and the incident response.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L222
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/ERC4626-Cloned.sol#L324

Tool used

Manual Review

Recommendation

Override the mint function and add the `whenNotPaused` modifier

Issue M-23: Buyouts of shorter duration liens can lead to the loss of borrower funds

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/171

Found by

0xRajeev

Summary

Liens whose duration is equal to (or maybe less than) `minDurationIncrease` cannot be bought out to be replaced by newer liens with lower interest rates but the same duration. This locks the borrower out of better-termed liens, effectively resulting in the loss of their funds

Vulnerability Detail

Liens whose duration is equal to (or maybe less than) `minDurationIncrease` cannot be bought out to be replaced by newer liens with lower interest rates but the exact duration because it results in an underflow in `_getRemainingInterest()`.

Example scenario: if the strategy`liendetails.duration` is <= 14 days, then it's impossible to do a buyout of a new lien because the implemented check requires to wait `minDurationIncrease`, which is set to 14 days. However, if the buyer waits 14 days, the lien is expired, which triggers the earlier mentioned underflow.

Impact

The borrower gets locked out of better-termed liens, effectively resulting in the loss of their funds because of extra interest paid on older liens.

Code Snippet

1. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L573
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L489-L490

Tool used

Manual Review

Recommendation

Revisit the checking logic and minimum duration as it applies to shorter-duration loans.

Discussion

SantiagoGregory

We updated buyoutLien() to check for a lower interest rate *or* higher duration.

Issue M-24: Loan duration can exceed the end of the next epoch

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/170

Found by

0xRajeev

Summary

Loan duration can exceed the end of the next epoch, which deviates from the protocol specification.

Vulnerability Detail

From the specs: "The duration of new loans is restricted to not exceed the end of the next epoch. For example, if a PublicVault is 15 days into a 30-day epoch, new loans must not be longer than 45 days."

However, there's no enforcement of this requirement.

Impact

The implementation does not adhere to the spec: Loan duration can exceed the end of the next epoch, which breaks protocol specification and therefore lead to miscalculations and potential fund loss.

Code Snippet

1. https://docs.astaria.xyz/docs/protocol-mechanics/epochs
2. https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/VaultImplementation.sol#L146-L228

Tool used

Manual Review

Recommendation

Implement as per specification or revisit the specification.

Issue M-25: `auctionExists()` is implemented in an unreliable way as an auction can exist but fail the check

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/164

Found by

yixxas

Summary

`auctionExists()` checks if an auction exist by checking if `auctions[tokenId].initiator!=address(0)`. This is problematic as an auction can be created with `auctions[tokenId].initiator=address(0)` which bypasses this check.

Vulnerability Detail

`auctionExists()` is used multiple times in both `CollateralToken.sol` and `LienToken.sol`. It is important that the function returns an accurate result. For example, `createLien()` can only be called if an auction exist, but in the case when `auctions[tokenId].initiator==address(0)`, this will revert even though auction is already created.

Impact

Multiple parts of the protocol will not be able to function due to a wrong return result from `auctionExists()`.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L67-L85 https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L318-L321

Tool used

Manual Review

Recommendation

To fix this, we can either ensure that `initiator` cannot be the 0 address, or we can use a more reliable check - `auctions[tokenId].firstBidTime!=0`, since `firstBidTime` is always set to `block.timestamp`.

However, I recommend that both of these are implemented as `initiator` should not be the 0 address as mentioned in my other issue #8.

Issue M-26: First ERC4626 deposit can break share calculation

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/143

Found by

pashov, rvierdiiev, neila, joestakey, ctf_sec, __141345__, Jeiwan, ak1, 0xNazgul

Summary

The first depositor of an ERC4626 vault can maliciously manipulate the share price by depositing the lowest possible amount (1 wei) of liquidity and then artificially inflating ERC4626.totalAssets.

This can inflate the base share price as high as 1:1e18 early on, which force all subsequence deposit to use this share price as a base and worst case, due to rounding down, if this malicious initial deposit front-run someone else depositing, this depositor will receive 0 shares and lost his deposited assets.

Vulnerability Detail

Given a vault with DAI as the underlying asset:

Alice (attacker) deposits initial liquidity of 1 wei DAI via `deposit()` Alice receives 1e18 (1 wei) vault shares Alice transfers 1 ether of DAI via transfer() to the vault to artificially inflate the asset balance without minting new shares. The asset balance is now 1 ether + 1 wei DAI -> vault share price is now very high (= 1000000000000000000001 wei ~ 1000 * 1e18) Bob (victim) deposits 100 ether DAI Bob receives 0 shares Bob receives 0 shares due to a precision issue. His deposited funds are lost.

The shares are calculated as following `returnsupply==0?assets:assets.mulDivDown(supply,totalAssets());` In case of a very high share price, due to totalAssets() > assets * supply, shares will be 0.

Impact

`ERC4626` vault share price can be maliciously inflated on the initial deposit, leading to the next depositor losing assets due to precision issues.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/ERC4626-Cloned.sol#L392

Tool used

Manual Review

Recommendation

This is a well-known issue, Uniswap and other protocols had similar issues when supply == 0.

For the first deposit, mint a fixed amount of shares, e.g. 10**decimals()

Issue M-27: LiquidityProvider can also lend to Private-Vault

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/141

Found by

neila

Summary

Insufficient access control for lending to PublicVault found by yawn-c111

Vulnerability Detail

Docs says as follows https://docs.astaria.xyz/docs/intro

> Any strategists may provide their own capital to fund these loans through their own `PrivateVaults`, and whitelisted strategists can deploy `PublicVaults` that accept funds from other liquidity providers.

However, Liquidity Providers can also lend to `PrivateVault`.

This is because `lendToVault` function is controlled by `mapping(address=>address)publicvaults`, which are managed by `_newVault` function and include `PrivateVaults`

This leads to unexpected atttack.

Impact

Unexpected liquidity providers can lend to private vaults

Code Snippet

https://github.com/unchain-dev/2022-10-astaria-UNCHAIN/blob/main/src/AstariaRouter.sol#L324

```
function lendToVault(IVault vault, uint256 amount) external whenNotPaused {
    TRANSFER_PROXY.tokenTransferFrom(
      address(WETH),
      address(msg.sender),
      address(this),
      amount
    );

    require(
      vaults[address(vault)] != address(0),
```

```
      "lendToVault: vault doesn't exist"
    );
    WETH.safeApprove(address(vault), amount);
    vault.deposit(amount, address(msg.sender));
  }
```

https://github.com/unchain-dev/2022-10-astaria-UNCHAIN/blob/main/src/AstariaRouter.sol#L500

```
function _newVault(
    uint256 epochLength,
    address delegate,
    uint256 vaultFee
  ) internal returns (address) {
    uint8 vaultType;

    address implementation;
    if (epochLength > uint256(0)) {
      require(
        epochLength >= minEpochLength && epochLength <= maxEpochLength,
        "epochLength must be greater than or equal to MIN_EPOCH_LENGTH and less
↪   than MAX_EPOCH_LENGTH"
      );
      implementation = VAULT_IMPLEMENTATION;
      vaultType = uint8(VaultType.PUBLIC);
    } else {
      implementation = SOLO_IMPLEMENTATION;
      vaultType = uint8(VaultType.SOLO);
    }

    //immutable data
    address vaultAddr = ClonesWithImmutableArgs.clone(
      implementation,
      abi.encodePacked(
        address(msg.sender),
        address(WETH),
        address(COLLATERAL_TOKEN),
        address(this),
        address(COLLATERAL_TOKEN.AUCTION_HOUSE()),
        block.timestamp,
        epochLength,
        vaultType,
        vaultFee
      )
    );

    //mutable data
```

```
    VaultImplementation(vaultAddr).init(
      VaultImplementation.InitParams(delegate)
    );

    vaults[vaultAddr] = msg.sender;

    emit NewVault(msg.sender, vaultAddr);

    return vaultAddr;
  }
```

Tool used

Manual Review

Recommendation

create requirement to lend to only PublicVaults.

Issue M-28: Can't call this function before epoch ended

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/136

Found by

neila

Summary

Can't call this function before epoch ended found by Tomosuke0930

Vulnerability Detail

The comment says as follows https://github.com/unchain-dev/2022-10-astaria-UNCHAIN/blob/main/src/PublicVault.sol#L236-L238

```
/**
  * @notice Rotate epoch boundary. This must be called before the next epoch can
 ↪  begin.
*/
```

However, this function can call the after `currentEpoch` ended

```
require(getEpochEnd(currentEpoch)  < block.timestamp, "Epoch has not ended");
```

Impact

If the `currentEpoch` is 3. `getEpochEnd(currentEpoch)` will be as follows.

```
START() + 4 * EPOCH_LENGTH()
```

And block.timestamp has to be bigger than this value to pass this check. Therefore, this function can call when the epoch enters into 4.

Code Snippet

https://github.com/unchain-dev/2022-10-astaria-UNCHAIN/blob/main/src/PublicVault.sol#L236-L249

```
/**
 * @notice Rotate epoch boundary. This must be called before the next epoch can
 ↪  begin.
 */
function processEpoch() external {
```

```
  // check to make sure epoch is over
  require(getEpochEnd(currentEpoch) < block.timestamp, "Epoch has not ended");
  require(withdrawReserve == 0, "Withdraw reserve not empty");
  if (liquidationAccountants[currentEpoch] != address(0)) {
    require(
      LiquidationAccountant(liquidationAccountants[currentEpoch])
        .getFinalAuctionEnd() < block.timestamp,
      "Final auction not ended"
    );
  }
```

https://github.com/unchain-dev/2022-10-astaria-UNCHAIN/blob/main/src/PublicVault.sol#L473-L475

```
function getEpochEnd(uint256 epoch) public view returns (uint256) {
    return START() + (epoch + 1) * EPOCH_LENGTH();
}
```

Tool used

Manual Review

Recommendation

Should change as follows to implement like the comment.

```
/**
 * @notice Rotate epoch boundary. This must be called before the next epoch can
 ↪  begin.
*/
function processEpoch() external {
 // check to make sure epoch is over
 require(getEpochEnd(currentEpoch - 1) < block.timestamp, "Epoch has not ended");
  /* ... */
}
```

Issue M-29: Users can't set the future epoch

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/134

Found by

neila

Summary

Users can't set the future epoch

Vulnerability Detail

Docs says as follows https://docs.astaria.xyz/docs/smart-contracts/WithdrawProxy

> liquidity providers must signal that they wish to withdraw funds at least one epoch in advance. However, `currentEpoch` is hardcoded

Impact

Users can't set future epoch for withdrawal

Code Snippet

https://github.com/unchain-dev/2022-10-astaria-UNCHAIN/blob/main/src/PublicVault.sol#L123

```
uint64 public currentEpoch = 0;
```

https://github.com/unchain-dev/2022-10-astaria-UNCHAIN/blob/main/src/PublicVault.sol#L161-L168

```
function withdraw(
    uint256 assets,
    address receiver,
    address owner
  ) public virtual override returns (uint256 shares) {
    shares = previewWithdraw(assets);
    redeemFutureEpoch(shares, receiver, owner, currentEpoch);
  }
```

https://github.com/unchain-dev/2022-10-astaria-UNCHAIN/blob/main/src/PublicVault.sol#L178-L199

```
function redeemFutureEpoch(
    uint256 shares,
    address receiver,
    address owner,
    uint64 epoch
  ) public virtual returns (uint256 assets) {
    // check to ensure that the requested epoch is not the current epoch or in
↪   the past
    // always epoch == currentEpoch by redeem

    require(epoch >= currentEpoch, "Exit epoch too low");
    require(msg.sender == owner, "Only the owner can redeem");
    // check for rounding error since we round down in previewRedeem.
```

Tool used

Manual Review

Recommendation

Let users to be able to set epoch

Issue M-30: LiquidationAccountant.claim function can be called any time

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/89

Found by

rvierdiiev

Summary

`LiquidationAccountant.claim` function can be called any time.

Vulnerability Detail

Function `LiquidationAccountant.claim` should not be called once auction is ongoing.

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LiquidationAccountant.sol#L66-L69

```
require(
  block.timestamp > finalAuctionEnd || finalAuctionEnd == uint256(0),
  "final auction has not ended"
);
```

However this check will be always true, because `finalAuctionEnd` is not a timestamp, but a period and you can't compare it with `block.timestamp`.

This is how this value is provided

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L407-L410

```
LiquidationAccountant(accountant).handleNewLiquidation(
  lien.amount,
  COLLATERAL_TOKEN.auctionWindow() + 1 days
);
```

And `COLLATERAL_TOKEN.auctionWindow()` is period.

Impact

The check will always be true, so you can call `claim` even auction is not ended yet.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

Use this in AstariaRouter when liquidate.

```
LiquidationAccountant(accountant).handleNewLiquidation(
  lien.amount,
  block.timestamp + COLLATERAL_TOKEN.auctionWindow() + 1 days
);
```

Issue M-31: Possible to fully block PublicVault.processEpoch function. No one will be able to receive their funds

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/88

Found by

TurnipBoy, rvierdiiev

Summary

Possible to fully block `PublicVault.processEpoch` function. No one will be able to receive their funds

Vulnerability Detail

When liquidity providers want to redeem their share from `PublicVault` they call `redeemFutureEpoch` function which will create new `WithdrawProxy` for the epoch(if not created already) and then mint shares for redeemer in `WithdrawProxy`. PublicVault transfer user's shares to himself.

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L178-L212

```
function redeemFutureEpoch(
  uint256 shares,
  address receiver,
  address owner,
  uint64 epoch
) public virtual returns (uint256 assets) {
  // check to ensure that the requested epoch is not the current epoch or in the
 ↪  past
  require(epoch >= currentEpoch, "Exit epoch too low");


  require(msg.sender == owner, "Only the owner can redeem");
  // check for rounding error since we round down in previewRedeem.


  ERC20(address(this)).safeTransferFrom(owner, address(this), shares);


  // Deploy WithdrawProxy if no WithdrawProxy exists for the specified epoch
  _deployWithdrawProxyIfNotDeployed(epoch);
```

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


  // WithdrawProxy shares are minted 1:1 with PublicVault shares
  WithdrawProxy(withdrawProxies[epoch]).mint(receiver, shares); // was
↪  withdrawProxies[withdrawEpoch]
}
```

This function mints `WithdrawProxy` shares 1:1 to redeemed `PublicVault` shares. Then later after call of `processEpoch` and `transferWithdrawReserve` the funds will be sent to the WithdrawProxy and users can now redeem their shares from it.

Function `processEpoch` decides how many funds should be sent to the `WithdrawProxy`.

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L268-L289

```
if (withdrawProxies[currentEpoch] != address(0)) {
  uint256 proxySupply = WithdrawProxy(withdrawProxies[currentEpoch])
    .totalSupply();


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


  if (liquidationAccountants[currentEpoch] != address(0)) {
    LiquidationAccountant(liquidationAccountants[currentEpoch])
      .setWithdrawRatio(liquidationWithdrawRatio);
  }


  uint256 withdrawAssets = convertToAssets(proxySupply);
  // compute the withdrawReserve
  uint256 withdrawLiquidations = liquidationsExpectedAtBoundary[
    currentEpoch
  ].mulDivDown(liquidationWithdrawRatio, 1e18);
  withdrawReserve = withdrawAssets - withdrawLiquidations;
  // burn the tokens of the LPs withdrawing
  _burn(address(this), proxySupply);


  _decreaseYIntercept(withdrawAssets);
}
```

This is how it is decided how much money should be sent to WithdrawProxy. Firstly, we look at totalSupply of WithdrawProxy. `uint256proxySupply=WithdrawProxy(withd`

rawProxies[currentEpoch]).totalSupply();.

And then we convert them to assets amount. uint256withdrawAssets=convertToAssets(proxySupply);

In the end function burns proxySupply amount of shares controlled by PublicVault. _burn(address(this),proxySupply);

Then this amount is allowed to be sent(if no auctions currently, but this is not important right now).

This all allows to attacker to make WithdrawProxy.deposit to mint new shares for him and increase totalSupply of WithdrawProxy, so proxySupply becomes more then was sent to PublicVault.

This is attack scenario.

1.PublicVault is created and funded with 50 ethers. 2.Someone calls redeemFutureEpoch function to create new WithdrawProxy for next epoch. 3.Attacker sends 1 wei to WithdrawProxy to make totalAssets be > 0. Attacker deposit to WithdrawProxy 1 wei. Now WithdrawProxy.totalSupply > PublicVault.balanceOf(PublicVault). 4.Someone call processEpoch and it reverts on burning.

As result, nothing will be send to WithdrawProxy where shares were minted for users. The just lost money.

Also this attack can be improved to drain users funds to attacker. Attacker should be liquidity provider. And he can initiate next redeem for next epoch, then deposit to new WithdrawProxy enough amount to get new shares. And call processEpoch which will send to the vault amount, that was not sent to previous attacked WithdrawProxy, as well. So attacker will take those funds.

Impact

Funds of PublicVault depositors are stolen.

Code Snippet

This is simple test that shows how external actor can corrupt WithdrawProxy.

```
function testWithdrawProxyDdos() public {
    Dummy721 nft = new Dummy721();
    address tokenContract = address(nft);
    uint256 tokenId = uint256(1);

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

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

    uint256 collateralId = tokenContract.computeId(tokenId);

    uint256 vaultTokenBalance = IERC20(publicVault).balanceOf(address(1));
    console.log("balance: ", vaultTokenBalance);

    // _signalWithdrawAtFutureEpoch(address(1), publicVault, uint64(1));
    _signalWithdraw(address(1), publicVault);

    address withdrawProxy = PublicVault(publicVault).withdrawProxies(
      PublicVault(publicVault).getCurrentEpoch()
    );

    vm.deal(address(2), 2);
    vm.startPrank(address(2));
      WETH9.deposit{value: 2}();
      //this we need to make share calculation not fail with divide by 0
      WETH9.transferFrom(address(2), withdrawProxy, 1);
      WETH9.approve(withdrawProxy, 1);

      //deposit 1 wei to make WithdrawProxy.totalSupply >
↪   PublicVault.balanceOf(address(PublicVault))
      //so processEpoch can't burn more shares
      WithdrawProxy(withdrawProxy).deposit(1, address(2));
    vm.stopPrank();

    assertEq(vaultTokenBalance, IERC20(withdrawProxy).balanceOf(address(1)));

    vm.warp(block.timestamp + 15 days);

    //processEpoch fails, because it can't burn more amount of shares, that was
↪   sent to PublicVault
    vm.expectRevert();
    PublicVault(publicVault).processEpoch();
  }
```

Tool used

Manual Review

Recommendation

Make function WithdrawProxy.deposit not callable.

Issue M-32: Vault: Unprotected mint / redeem function

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/84

Found by

0x4141

Summary

`Vault` only protects the `deposit` / `withdraw` functions.

Vulnerability Detail

Within `Vault`, the `deposit` / `withdraw` functions are only callable by the owner. However, because the contracts inherits from ERC4626, there are also `mint` and `redeem` functions which are callable by anyone.

Impact

Anyone can fund and exit the vault, which should not be possible for private vaults.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L85

Tool used

Manual Review

Recommendation

Also overwrite the `mint` and `redeem` functions.

Issue M-33: LiquidationAccountant.claim may revert for some tokens

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/83

Found by

0x4141

Summary

`LiquidationAccountant.claim` may initiate a transfer with the amount 0, which reverts for some tokens.

Vulnerability Detail

Some tokens (e.g., LEND -> see https://github.com/d-xo/weird-erc20#revert-on-zero-value-transfers) revert when a transfer with amount 0 is initiated. This can happen within `claim` when the `withdrawRatio` is 100%.

Impact

In such a scenario, the funds are not claimable, leading to a loss of funds.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LiquidationAccountant.sol#L88

Tool used

Manual Review

Recommendation

Do not initiate a transfer when the amount is zero.

Issue M-34: LienToken._payment function increases users debt

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/73

Found by

rvierdiiev

Summary

LienToken._payment function increases users debt by setting `lien.amount=_getOwed(lien)`

Vulnerability Detail

`LienToken._payment` is used by `LienToken.makePayment` function that allows borrower to repay part or all his debt.

Also this function can be called by `AuctionHouse` when the lien is liquidated.

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L594-L649

```
function _payment(
  uint256 collateralId,
  uint8 position,
  uint256 paymentAmount,
  address payer
) internal returns (uint256) {
  if (paymentAmount == uint256(0)) {
    return uint256(0);
  }


  uint256 lienId = liens[collateralId][position];
  Lien storage lien = lienData[lienId];
  uint256 end = (lien.start + lien.duration);
  require(
    block.timestamp < end || address(msg.sender) == address(AUCTION_HOUSE),
    "cannot pay off an expired lien"
  );


  address lienOwner = ownerOf(lienId);
  bool isPublicVault = IPublicVault(lienOwner).supportsInterface(
    type(IPublicVault).interfaceId
```

```
    );


    lien.amount = _getOwed(lien);


    address payee = getPayee(lienId);
    if (isPublicVault) {
      IPublicVault(lienOwner).beforePayment(lienId, paymentAmount);
    }
    if (lien.amount > paymentAmount) {
      lien.amount -= paymentAmount;
      lien.last = block.timestamp.safeCastTo32();
      // slope does not need to be updated if paying off the rest, since we
↪     neutralize slope in beforePayment()
      if (isPublicVault) {
        IPublicVault(lienOwner).afterPayment(lienId);
      }
    } else {
      if (isPublicVault && !AUCTION_HOUSE.auctionExists(collateralId)) {
        // since the openLiens count is only positive when there are liens that
↪     haven't been paid off
        // that should be liquidated, this lien should not be counted anymore
        IPublicVault(lienOwner).decreaseEpochLienCount(
          IPublicVault(lienOwner).getLienEpoch(end)
        );
      }
      //delete liens
      _deleteLienPosition(collateralId, position);
      delete lienData[lienId]; //full delete


      _burn(lienId);
    }


    TRANSFER_PROXY.tokenTransferFrom(WETH, payer, payee, paymentAmount);


    emit Payment(lienId, paymentAmount);
    return paymentAmount;
  }
```

The main problem is in line 617. `lien.amount=_getOwed(lien);` https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L617

Here lien.amount becomes lien.amount + accrued interests, because `_getOwed` do that calculation.

`lien.amount` is the amount that user borrowed. So actually that line has just increased user's debt. And in case if he didn't pay all amount of lien, then next time he will pay more interests.

Example. User borrows 1 eth. His `lien.amount` is 1eth. Then he wants to repay some part(let's say 0.5 eth). Now his `lien.amount` becomes `lien.amount+interests`. When he pays next time, he pays `(lien.amount+interests)+newinterests`. So interests are acummulated on previous interests.

Impact

User borrowed amount increases and leads to lose of funds.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

Do not update lien.amount to _getOwed(lien).

Discussion

Evert0x

Downgrading to medium

Issue M-35: Any public vault without a delegate can be drained

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/69

Found by

obront, HonorLt, cryptphi, zzykxx, yixxas, rvierdiiev

Summary

If a public vault is created without a delegate, delegate will have the value of `address(0)`. This is also the value returned by `ecrecover` for invalid signatures (for example, if v is set to a position number that is not 27 or 28), which allows a malicious actor to cause the signature validation to pass for arbitrary parameters, allowing them to drain a vault using a worthless NFT as collateral.

Vulnerability Detail

When a new Public Vault is created, the Router calls the `init()` function on the vault as follows:

```
VaultImplementation(vaultAddr).init(
  VaultImplementation.InitParams(delegate)
);
```

If a delegate wasn't set, this will pass `address(0)` to the vault. If this value is passed, the vault simply skips the assignment, keeping the delegate variable set to the default 0 value:

```
if (params.delegate != address(0)) {
  delegate = params.delegate;
}
```

Once the delegate is set to the zero address, any commitment can be validated, even if the signature is incorrect. This is because of a quirk in `ecrecover` which returns `address(0)` for invalid signatures. A signature can be made invalid by providing a positive integer that is not 27 or 28 as the `v` value. The result is that the following function call assigns `recovered=address(0)`:

```
address recovered = ecrecover(
  keccak256(
    encodeStrategyData(
      params.lienRequest.strategy,
      params.lienRequest.merkle.root
```

```
    )
  ),
  params.lienRequest.v,
  params.lienRequest.r,
  params.lienRequest.s
);
```

To confirm the validity of the signature, the function performs two checks:

```
require(
  recovered == params.lienRequest.strategy.strategist,
  "strategist must match signature"
);
require(
  recovered == owner() || recovered == delegate,
  "invalid strategist"
);
```

These can be easily passed by setting the `strategist` in the params to `address(0)`. At this point, all checks will pass and the parameters will be accepted as approved by the vault.

With this power, a borrower can create params that allow them to borrow the vault's full funds in exchange for a worthless NFT, allowing them to drain the vault and steal all the user's funds.

Impact

All user's funds held in a vault with no delegate set can be stolen.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L537-L539

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/VaultImplementation.sol#L118-L124

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/VaultImplementation.sol#L167-L185

Tool used

Manual Review, Foundry

Recommendation

Add a require statement that the recovered address cannot be the zero address:

```
require(recovered != address(0));
```

Issue M-36: Strategist nonce is not checked

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/59

Found by

ctf_sec, cryptphi, rvierdiiev, HonorLt

Summary

Strategist nonce is not checked while checking commitment. This makes impossible for strategist to cancel signed commitment.

Vulnerability Detail

`VaultImplementation.commitToLien` is created to give the ability to borrow from the vault. The conditions of loan are discussed off chain and owner or delegate of the vault then creates and signes deal details. Later borrower can provide it as `IAstariaRouter.Commitmentcalldataparams` param to VaultImplementation.commitToLien.

After the checking of signer of commitment `VaultImplementation._validateCommitment` function calls `AstariaRouter.validateCommitment`.

```
function validateCommitment(IAstariaRouter.Commitment calldata commitment)
  public
  returns (bool valid, IAstariaRouter.LienDetails memory ld)
{
  require(
    commitment.lienRequest.strategy.deadline >= block.timestamp,
    "deadline passed"
  );


  require(
    strategyValidators[commitment.lienRequest.nlrType] != address(0),
    "invalid strategy type"
  );


  bytes32 leaf;
  (leaf, ld) = IStrategyValidator(
    strategyValidators[commitment.lienRequest.nlrType]
  ).validateAndParse(
      commitment.lienRequest,
      COLLATERAL_TOKEN.ownerOf(
        commitment.tokenContract.computeId(commitment.tokenId)
      ),
```

```
      commitment.tokenContract,
      commitment.tokenId
    );


  return (
    MerkleProof.verifyCalldata(
      commitment.lienRequest.merkle.proof,
      commitment.lienRequest.merkle.root,
      leaf
    ),
    ld
  );
}
```

This function check additional params, one of which is `commitment.lienRequest.strategy.deadline`. But it doesn't check for the nonce of strategist here. But this nonce is used while signing.

Also `AstariaRouter` gives ability to increment nonce for strategist, but it is never called. That means that currently strategist use always same nonce and can't cancel his commitment.

Impact

Strategist can't cancel his commitment. User can use this commitment to borrow up to 5 times.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

Give ability to strategist to call `increaseNonce` function.

Issue M-37: _validateCommitment fails for approved operators

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/52

Found by

obront, rvierdiiev

Summary

If a collateral token owner approves another user as an operator for all their tokens (rather than just for a given token), the validation check in `_validateCommitment()` will fail.

Vulnerability Detail

The collateral token is implemented as an ERC721, which has two ways to approve another user:

- Approve them to take actions with a given token (`approve()`)
- Approve them as an "operator" for all your owned tokens (`setApprovalForAll()`)

However, when the `_validateCommitment()` function checks that the token is owned or approved by `msg.sender`, it does not accept those who are set as operators.

```
if (msg.sender != holder) {
  require(msg.sender == operator, "invalid request");
}
```

Impact

Approved operators of collateral tokens will be rejected from taking actions with those tokens.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/VaultImplementation.sol#L152-L158

Tool used

Manual Review

Recommendation

Include an additional check to confirm whether the `msg.sender` is approved as an operator on the token:

```
address holder = ERC721(COLLATERAL_TOKEN()).ownerOf(collateralId);
address approved = ERC721(COLLATERAL_TOKEN()).getApproved(collateralId);
address operator = ERC721(COLLATERAL_TOKEN()).isApprovedForAll(holder);

if (msg.sender != holder) {
  require(msg.sender == operator || msg.sender == approved, "invalid request");
}
```

Issue M-38: timeToEpochEnd calculates backwards, breaking protocol math

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/50

Found by

obront, hansfriese, 0xRajeev, 0xNazgul

Summary

When a lien is liquidated, it calls `timeToEpochEnd()` to determine if a liquidation accountant should be deployed and we should adjust the protocol math to expect payment in a future epoch. Because of an error in the implementation, all liquidations that will pay out in the current epoch are set up as future epoch liquidations.

Vulnerability Detail

The `liquidate()` function performs the following check to determine if it should set up the liquidation to be paid out in a future epoch:

```
if (PublicVault(owner).timeToEpochEnd() <= COLLATERAL_TOKEN.auctionWindow())
```

This check expects that `timeToEpochEnd()` will return the time until the epoch is over. However, the implementation gets this backwards:

```
function timeToEpochEnd() public view returns (uint256) {
  uint256 epochEnd = START() + ((currentEpoch + 1) * EPOCH_LENGTH());

  if (epochEnd >= block.timestamp) {
    return uint256(0);
  }

  return block.timestamp - epochEnd;
}
```

If `epochEnd>=block.timestamp`, that means that there IS remaining time in the epoch, and it should perform the calculation to return `epochEnd-block.timestamp`. In the opposite case, where `epochEnd<=block.timestamp`, it should return zero.

The result is that the function returns 0 for any epoch that isn't over. Since `0<COLLATERAL_TOKEN.auctionWindow())`, all liquidated liens will trigger a liquidation accountant and the rest of the accounting for future epoch withdrawals.

Impact

Accounting for a future epoch withdrawal causes a number of inconsistencies in the protocol's math, the impact of which vary depending on the situation. As a few examples:

- It calls `decreaseEpochLienCount()`. This has the effect of artificially lowering the number of liens in the epoch, which will cause the final liens paid off in the epoch to revert (and will let us process the epoch earlier than intended).
- It sets the payee of the lien to the liquidation accountant, which will pay out according to the withdrawal ratio (whereas all funds should be staying in the vault).
- It calls `increaseLiquidationsExpectedAtBoundary()`, which can throw off the math when processing the epoch.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L562-L570

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L388-L415

Tool used

Manual Review

Recommendation

Fix the `timeToEpochEnd()` function so it calculates the remaining time properly:

```
function timeToEpochEnd() public view returns (uint256) {
  uint256 epochEnd = START() + ((currentEpoch + 1) * EPOCH_LENGTH());

  if (epochEnd <= block.timestamp) {
    return uint256(0);
  }

  return epochEnd - block.timestamp; //
}
```

Issue M-39: Underlying With Non-Standard Decimals Not Supported

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/33

Found by

0x0

Summary

Arithmetic operations are performed with the assumption that the token always has 18 decimals.

Vulnerability Detail

LiquidationAccountant.claim

Arithmetic operations assume the token has 18 decimals. Not all tokens use 18 decimals, such as Tether.

Impact

- The addition of underlying capital that does not use 18 decimals will not be possible.

Code Snippet

```
uint256 transferAmount = withdrawRatio.mulDivDown(balance, 1e18);
```

Tool used

Manual Review

Recommendation

- Consider whether the addition of capital that does not use 18 decimals is desirable in the future. If it is, refactor contracts to support tokens with non-standard decimals.

Issue M-40: _payment() function transfers full paymentAmount, overpaying first liens

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/28

Found by

obront, 0xRajeev, bin2chen, hansfriese, tives, __141345__, ak1, rvierdiiev

Summary

The `_payment()` function sends the full `paymentAmount` argument to the lien owner, which both (a) overpays lien owners if borrowers accidentally overpay and (b) sends the first lien owner all the funds for the entire loop of a borrower is intending to pay back multiple loans.

Vulnerability Detail

There are two `makePayment()` functions in LienToken.sol. One that allows the user to specific a `position` (which specific lien they want to pay back, and another that iterates through their liens, paying each back.

In both cases, the functions call out to `_payment()` with a `paymentAmount`, which is sent (in full) to the lien owner.

```
TRANSFER_PROXY.tokenTransferFrom(WETH, payer, payee, paymentAmount);
```

This behavior can cause problems in both cases.

The first case is less severe: If the user is intending to pay off one lien, and they enter a `paymentAmount` greater than the amount owed, the function will send the full `paymentAmount` to the lien owner, rather than just sending the amount owed.

The second case is much more severe: If the user is intending to pay towards all their loans, the `_makePayment()` function loops through open liens and performs the following:

```
uint256 paymentAmount = totalCapitalAvailable;
for (uint256 i = 0; i < openLiens.length; ++i) {
  uint256 capitalSpent = _payment(
    collateralId,
    uint8(i),
    paymentAmount,
    address(msg.sender)
  );
```

```
    paymentAmount -= capitalSpent;
}
```

The `_payment()` function is called with the first lien with `paymentAmount` set to the full amount sent to the function. The result is that this full amount is sent to the first lien holder, which could greatly exceed the amount they are owed.

Impact

A user who is intending to pay off all their loans will end up paying all the funds they offered, but only paying off their first lien, potentially losing a large amount of funds.

Code Snippet

The `_payment()` function with the core error:

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L594-L649

The `_makePayment()` function that uses `_payment()` and would cause the most severe harm:

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L410-L424

Tool used

Manual Review

Recommendation

In `_payment()`, if `lien.amount<paymentAmount`, set `paymentAmount=lien.amount`.

The result will be that, in this case, only `lien.amount` is transferred to the lien owner, and this value is also returned from the function to accurately represent the amount that was paid.

Issue M-41: _getInterest() function uses block.timestamp instead of the inputted timestamp

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/25

Found by

obront, sorrynotsorry

Summary

The `_getInterest()` function takes a timestamp as input. However, in a crucial check in the function, it uses `block.timestamp` instead. The result is that other functions expecting accurate interest amounts will receive incorrect values.

Vulnerability Detail

The `_getInterest()` function takes a lien and a timestamp as input. The intention is for it to calculate the amount of time that has passed in the lien (`delta_t`) and multiply this value by the rate and the amount to get the interest generated by this timestamp.

However, the function uses the following check regarding the timestamp:

```
if (block.timestamp >= lien.start + lien.duration) {
  delta_t = uint256(lien.start + lien.duration - lien.last);
}
```

Because this check uses `block.timestamp` before returning the maximum interest payment, the function will incorrectly determine which path to take, and return an incorrect interest value.

Impact

There are two negative consequences that can come from this miscalculation:

- if the function is called when the lien is over (`block.timestamp>=lien.start+lien.duration`) to check an interest amount from a timestamp during the lien, it will incorrectly return the maximum interest value
- If the function is called when the lien is active for a timestamp long after the lien is over, it will skip the check to return maximum value and return the value that would have been generated if interest kept accruing indefinitely (using `delta_t=uint256(timestamp.safeCastTo32()-lien.last);`)

This `_getInterest()` function is used in many crucial protocol functions (`_getOwed()`, `calculateSlope()`, `changeInSlope()`, `getTotalDebtForCollateralToken()`), so these incorrect values can have surprising and unexpected negative impacts on the protocol.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/LienToken.sol#L177-L196

Tool used

Manual Review

Recommendation

Change `block.timestamp` to `timestamp` so that the if statement checks correctly.

Issue M-42: Checks for epoch lengths of new vaults are slightly off

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/23

Found by

obront

Summary

The check that new vaults has an `epochLength` in the range from `minEpochLength` (inclusive) to `maxEpochLength` (not inclusive) is slightly off.

Vulnerability Detail

The error string for an incorrectly set epoch length reads:

> "epochLength must be greater than or equal to MIN_EPOCH_LENGTH and less than MAX_EPOCH_LENGTH"

However, in the code implementing this check, an `epochLength` equal to `maxEpochLength` is allowed.

```
require(
  epochLength >= minEpochLength && epochLength <= maxEpochLength,
  "epochLength must be greater than or equal to MIN_EPOCH_LENGTH and less than
↪   MAX_EPOCH_LENGTH"
);
```

Impact

Slightly longer epochs will be permitted than was intended.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/AstariaRouter.sol#L509-L512

Tool used

Manual Review

Recommendation

Change the check in the require statement to match the intended behavior:

```
require(
  epochLength >= minEpochLength && epochLength < maxEpochLength,
  "epochLength must be greater than or equal to MIN_EPOCH_LENGTH and less than
  ↪  MAX_EPOCH_LENGTH"
);
```

Issue M-43: Vault Fee uses incorrect offset leading to wildly incorrect value, allowing strategists to steal all funds

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/19

Found by

obront, zzykxx

Summary

`VAULT_FEE()` uses an incorrect offset, returning a number ~1e16X greater than intended, providing strategists with unlimited access to drain all vault funds.

Vulnerability Detail

When using ClonesWithImmutableArgs, offset values are set so that functions representing variables can retrieve the correct values from storage.

In the ERC4626-Cloned.sol implementation, `VAULT_TYPE()` is given an offset of 172. However, the value before it is a `uint8` at the offset 164. Since a `uint8` takes only 1 byte of space, `VAULT_TYPE()` should have an offset of 165.

I put together a POC to grab the value of `VAULT_FEE()` in the test setup:

```
function testVaultFeeIncorrectlySet() public {
  Dummy721 nft = new Dummy721();
  address tokenContract = address(nft);
  uint256 tokenId = uint256(1);
  address publicVault = _createPublicVault({
      strategist: strategistOne,
      delegate: strategistTwo,
      epochLength: 14 days
  });
  uint fee = PublicVault(publicVault).VAULT_FEE();
  console.log(fee)
  assert(fee == 5000); // 5000 is the value that was meant to be set
}
```

In this case, the value returned is > 3e20.

Impact

This is a highly critical bug. `VAULT_FEE()` is used in `_handleStrategistInterestReward()` to determine the amount of tokens that should be allocated to `strategistUnclaimedShares`.

```
if (VAULT_FEE() != uint256(0)) {
    uint256 interestOwing = LIEN_TOKEN().getInterest(lienId);
    uint256 x = (amount > interestOwing) ? interestOwing : amount;
    uint256 fee = x.mulDivDown(VAULT_FEE(), 1000); //VAULT_FEE is a basis point
    strategistUnclaimedShares += convertToShares(fee);
  }
```

The result is that strategistUnclaimedShares will be billions of times higher than the total interest generated, essentially giving strategist access to withdraw all funds from their vaults at any time.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/ERC4626-Cloned.sol#L113-L116

https://github.com/sherlock-audit/2022-10-astaria/blob/main/src/PublicVault.sol#L518-L523

Tool used

Manual Review

Recommendation

Set the offset for `VAULT_FEE()` to 165. I tested this value in the POC I created and it correctly returned the value of 5000.

Issue M-44: Bids cannot be created within timeBuffer of completion of a max duration auction

Source: https://github.com/sherlock-audit/2022-10-astaria-judging/issues/18

Found by

obront, 0xRajeev, Prefix, neila, csanuragjain, hansfriese, 0x4141, TurnipBoy, minhquanym, Jeiwan, yixxas, rvierdiiev, peanuts

Summary

The auction mechanism is intended to watch for bids within `timeBuffer` of the end of the auction, and automatically increase the remaining duration to `timeBuffer` if such a bid comes in.

There is an error in the implementation that causes all bids within `timeBuffer` of the end of a max duration auction to revert, effectively ending the auction early and cutting off bidders who intended to wait until the end.

Vulnerability Detail

In the `createBid()` function in AuctionHouse.sol, the function checks if a bid is within the final `timeBuffer` of the auction:

```
if (firstBidTime + duration - block.timestamp < timeBuffer)
```

If so, it sets `newDuration` to equal the amount that will extend the auction to `timeBuffer` from now:

```
uint64 newDuration = uint256( duration + (block.timestamp + timeBuffer -
↪  firstBidTime) ).safeCastTo64();
```

If this `newDuration` doesn't extend beyond the `maxDuration`, this works great. However, if it does extend beyond `maxDuration`, the following code is used to update `duration`:

```
auctions[tokenId].duration = auctions[tokenId].maxDuration - firstBidTime;
```

This code is incorrect. `maxDuration` will be a duration for the contest (currently set to 3 days), whereas `firstTimeBid` is a timestamp for the start of the auction (current timestamps are > 1 billion).

Subtracting `firstTimeBid` from `maxDuration` will underflow, which will revert the function.

Impact

- Bidders who expected to wait until the end of the auction to vote will be cut off from voting, as the auction will revert their bids.
- Vaults whose collateral is up for auction will earn less than they otherwise would have.

Code Snippet

https://github.com/sherlock-audit/2022-10-astaria/blob/main/lib/astaria-gpl/src/AuctionHouse.sol#L127-L146

Tool used

Manual Review

Recommendation

Change this assignment to simply assign `duration` to `maxDuration`, as follows:

```
auctions[tokenId].duration = auctions[tokenId].maxDuration
```