apDAO Security Review

Pashov Audit Group

Conducted by: samuraii77, 0xbepresent, yttriumzz

October 3rd - October 12th

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **0xHoneyJar/apdao-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About apDAO

Apiology DAO is implementing a custom ERC20DAO contract alongside a specialized auction house for seat tokens, enabling users to auction and bid within a decentralized structure. The integration includes governance and seat tokens that align with auction functionality, ensuring secure and non-transferable ownership regulated through custom minting and burning conditions.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

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

6. Security Assessment Summary

review commit hash - 225065a80c717e1bd6129ae7fd6652e922270b2d

fixes review commit hash - a1d5172b8fe2d80be6db01bd65b2394259a8b00d

Scope

The following smart contracts were in scope of the audit:

- `Deployer`
- `emitter`
- `erc20dao`
- `factory`
- `helper`
- `proxy`
- `ApiologyDAOAuctionHouse`
- `ApiologyDAOToken`

7. Executive Summary

Over the course of the security review, samuraii77, 0xbepresent, yttriumzz engaged with apDAO to review apDAO. In this period of time a total of **35** issues were uncovered.

Protocol Summary

Protocol Name	apDAO
Repository	https://github.com/0xHoneyJar/apdao-contracts
Date	October 3rd - October 12th
Protocol Type	DAO

Findings Count

Severity	**Amount**
Critical	2
High	5
Medium	8
Low	20
Total Findings	**35**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Frontrunning the auction creation and settlement to lower its listed price or to grief the NFT owner	Critical	Resolved
[C-02]	The winner of a bid can brick the auction house	Critical	Resolved
[H-01]	Users can manipulate the NFT chosen for the auction	High	Resolved
[H-02]	The attacker can exit the queue after requesting a random	High	Resolved
[H-03]	Incorrect usage of msg.value in iterative native token distribution	High	Resolved
[H-04]	Reversion in ApiologyDAOToken transfer due to insufficient governance tokens	High	Resolved
[H-05]	Potential overwriting of auction storage leading to loss of auctioned tokens	High	Resolved
[M-01]	An auction created when there is no treasury backing	Medium	Resolved
[M-02]	mintToken() assumes that the user we mint to is not an admin	Medium	Resolved
[M-03]	setPrices() accesses the settlement history by a wrong value	Medium	Resolved
[M-04]	Auction creation during paused state due to entropy callback	Medium	Resolved
[M-05]	Failure of entropy source callback	Medium	Resolved

[M-06]	Unhandled exceptions in a function	Medium	Resolved
[M-07]	createERC20DAO() will always revert when _depositChainIds has a length bigger than 1	Medium	Acknowledged
[M-08]	Bias in random token selection due to modulo operator	Medium	Acknowledged
[L-01]	Mismatch between checks for storage variables	Low	Resolved
[L-02]	Difference between emitted events	Low	Acknowledged
[L-03]	Not checking of being done upon the ERC20 DAO creation	Low	Resolved
[L-04]	updateDepositTime() allows for setting the deposit close time in the past	Low	Acknowledged
[L-05]	Mismatch between the minimum and maximum deposits	Low	Resolved
[L-06]	Multiple instances of wrongly assuming a settlement state is empty	Low	Resolved
[L-07]	The factory.emitSignerChanged function allows any dao address	Low	Acknowledged
[L-08]	The factory.createERC20DAO function may deploy an empty Gnosis Safe contract	Low	Acknowledged
[L-09]	The factory does not check whether the create fees are paid successfully	Low	Resolved
[L-10]	Losing funds because the contract forces the use of WETH when ETH transfer fails	Low	Acknowledged
[L-11]	Absence of event emissions in critical functions	Low	Resolved

[L-12]	Redundant role assignment	Low	Resolved
[L-13]	Unnecessary use of payable modifier	Low	Acknowledged
[L-14]	Unnecessary random number request with single NFT in queue	Low	Resolved
[L-15]	Redundant gnosis safe deployment in factory::createERC20DAO	Low	Acknowledged
[L-16]	Insufficient validation of maxDepositPerUser and pricePerToken	Low	Acknowledged
[L-17]	Absence of the payable modifier for request fees	Low	Acknowledged
[L-18]	createERC20DAO() uses the deposit time input in a wrong way	Low	Resolved
[L-19]	Malicious bidders can disincentivize users from bidding	Low	Acknowledged
[L-20]	Attackers can frontrun and increase refund gas to DoS the bid	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Frontrunning the auction creation and settlement to lower its listed price or to grief the NFT owner

Severity

Impact: High

Likelihood: High

Description

Upon an auction creation, we have this line:

```
uint256 currentRFV = treasury.realFloorValue();
```

This is the price an auctioned NFT is listed for (increased by a percentage serving as a buffer). This is how that value is computed:

```
value_ = (backing_ + backingLoanedOut_) / float_;
```

The `float_` variable is the circulating supply of NFTs that are not owned by the treasury. If we see the calculation, we can clearly see that a higher `float_` value will result in a lower result. Thus, any user (or many users) can frontrun the auction creation and call `ApiologyDAOToken::claim()` to mint themselves NFTs which will result in a higher supply or a higher `float_` value:

```
_mint(msg.sender, remainingClaimable);
```

This will result in the starting price of the auction being lower than expected.

A similar thing can happen when there have been no bidders for an NFT and the auction is being settled (code for calculating the value is in `Treasury::redeemItem()`):

```
uint256 redeemedValue = treasury.redeemItem(_auction.apdaoId);

                // Convert WETH to ETH
                IWETH(weth).withdraw(redeemedValue);

                // Transfer the exact redeemed ETH value to the original owner
                _safeTransferETHWithFallback(originalOwner, redeemedValue);
```

This will cause the NFT owner who put his NFT on an auction to receive a significantly lower amount of funds than expected and than the actual reserved price.

Recommendations

The possible fix is disallowing claiming in the time delta between the call of `_createAuction()` and settling the claim however that would still leave that vector possible when the auction queue is empty as then the auction is directly created without first requesting a random number.

[C-02] The winner of a bid can brick the auction house

Severity

Impact: High

Likelihood: High

Description

Whenever an auction is over, `_settleAuction()` is called. We have 2 main flows there - when no one has bid for the auction and when there is a bidder for the auction. We will focus on the latter. One of the lines we have for that flow is this one:

```
apiologyToken.safeTransferFrom(address
  (this), _auction.bidder, _auction.apdaoId);
```

It uses `safeTransferFrom()` to transfer the NFT from the contract to the bidder. Thus, if the bidder is a contract that either maliciously or accidentally doesn't properly implement the `onERC721Received()` function, then it will lead to a revert leading to the auction house being bricked. The NFT will be stuck as well as the funds paid by the bidder for it.

Recommendations

Consider using `transferFrom()` instead of `safeTransferFrom()`

8.2. High Findings

[H-01] Users can manipulate the NFT chosen for the auction

Severity

Impact: High

Likelihood: Medium

Description

To create an auction, a random number is generated which is used to determine the index of the NFT auctioned off from the `auctionQueue`. If the random number is equal to the length of the auction queue or if the random number is 1 less than the length of the auction queue, users can manipulate the NFT chosen in their favor.

Firstly, let's imagine the first scenario:

- An auction will be created with the random number 5 and there are 5 NFTs in the auction queue
- Due to the line below, we will auction off the NFT with index of 0:

```
uint256 randomIndex = uint256(randomNumber) % auctionQueue.length;
```

- However, a user frontruns the auction creation and puts his NFT in the auction queue causing the index of the NFT to instead equal `5 % 6 = 5` which is his NFT

Now, let's imagine the other scenario:

- An auction will be created with the random number of 5 and there are 6 NFTs in the auction queue
- Due to the line above, the index of the to-be-auctioned NFT will be `5 % 6 = 5` or in other words, the last NFT will be auctioned off
- However, a user who has already put his NFT in the queue frontruns the auction creation and does the following:
  - Take out his NFT from the queue
  - Put it back in

- Now the NFT has gone from slot `x` in the queue to the last spot, the index of the to-be-auctioned NFT will still be the same which results in the user who conducted the sequence above, being the owner of the auctioned NFT.

Recommendations

As the request and the actual receipt of the random number are in separate transactions, consider disallowing adding and removing from the auction queue whenever `_createAuction()` is called.

[H-02] The attacker can exit the queue after requesting a random

Severity

Impact: High

Likelihood: Medium

Description

There are two situations when the `ApiologyAuctionHouse` contract creates an auction:

1. When there are seats in the `auctionQueue`, it will request a random number from Pyth. After Pyth generates the random number, it will call back the `entropyCallback` function to create the auction. The specific process is as follows.
   1. (tx1) User calls `ApiologyAuctionHouse.settleCurrentAndCreateNewAuction`

    2. (tx1) `ApiologyAuctionHouse` requests random numbers from Pyth and pay some fees to Pyth
    3. (off-chain) Pyth generates random numbers
    4. (tx2) Pyth calls `ApiologyAuctionHouse.entropyCallback` create the auction
2. When there are no seats in the `auctionQueue`, the auction is generated directly.

The problem exists in the first scenario. The attacker can put the NFT into the queue first, and then remove the NFT from the queue before Pyth generates the random number. This results in that every time `settleCurrentAndCreateNewAuction` is called, it will request a random number from Pyth, wasting the contract's funds and then creating an empty auction.

An attacker can use an NFT to repeatedly trigger this bug and consume the funds in the contract.

Recommendations

It is recommended that users who exit the queue pay a fee or that the user not exit the queue before receiving the random number.

[H-03] Incorrect usage of `msg.value` in iterative native token distribution

Severity

Impact: Medium

Likelihood: High

Description

In the `factory::createERC20DAO` function, when the `_depositChainIds` array has more than one element, the function iterates over the chain IDs to send messages using the `commLayer::sendMsg` function. A portion of the native token value (`msg.value`) is calculated and sent in each iteration using the formula `(msg.value - fees) / (_depositChainIds.length - (i + 1))`. However, the calculation does not account for the diminishing `msg.value` as

fees are deducted in each iteration. This might lead to incorrect fee distribution for the message transfers and possibly result in insufficient funds for subsequent message transfers.

```
File: factory.sol
243:         if (_depositChainIds.length > 1) {
244:             for (uint256 i; i < _depositChainIds.length - 1;) {
245:                 require
  (_depositTokenAddress[i + 1] != NATIVE_TOKEN_ADDRESS, "Native not supported");
246:                 bytes memory _payload = abi.encode(
247:                     _commLayerId,
248:                     _distributionAmount,
249:                     amountToSD(_depositTokenAddress[0], _pricePerToken),
250:                     amountToSD
  (_depositTokenAddress[0], _minDepositPerUser),
251:                     amountToSD
  (_depositTokenAddress[0], _maxDepositPerUser),
252:                     _ownerFeePerDepositPercent,
253:                     _depositTime,
254:                     _quorumPercent,
255:                     _thresholdPercent,
256:                     _safeThreshold,
257:                     _depositChainIds,
258:                     _daoAddress,
259:                     _depositTokenAddress[i + 1],
260:                     _admins,
261:                     _onlyAllowWhitelist,
262:                     _merkleRoot,
263:                     0,
264:                     msg.sender
265:                 );
266:>>>              ICommLayer(commLayer).sendMsg{value: (msg.value - fees) /
  (_depositChainIds.length - (i + 1))}(
267:>>>                  commLayer, _payload, abi.encode
  (_depositChainIds[i + 1], msg.sender)
268:                 );
269:                 unchecked {
270:                     ++i;
271:                 }
272:             }
273:
274:             IEmitter(emitterAddress).createCCDao
  (_daoAddress, _depositChainIds);
275:         }
```

Each call to `sendMsg` should account for the portion of `msg.value` that has already been distributed in previous iterations. Failing to do so can result in insufficient funds being sent for each subsequent message, leading to failures or incomplete operations.

Recommendations

Update the calculation to account for the funds that have already been allocated in previous iterations. This ensures the correct distribution of remaining funds for each subsequent message.

```
+           uint256 valueToDistribute = msg.value - fees;
            for (uint256 i; i < _depositChainIds.length - 1;) {
                require(
                  _depositTokenAddress[i+1]!=NATIVE_TOKEN_ADDRESS,
                  "Nativenotsupported"
                );
                bytes memory _payload = abi.encode(
                    _commLayerId,
                    _distributionAmount,
                    amountToSD(_depositTokenAddress[0], _pricePerToken),
                    amountToSD(_depositTokenAddress[0], _minDepositPerUser),
                    amountToSD(_depositTokenAddress[0], _maxDepositPerUser),
                    _ownerFeePerDepositPercent,
                    _depositTime,
                    _quorumPercent,
                    _thresholdPercent,
                    _safeThreshold,
                    _depositChainIds,
                    _daoAddress,
                    _depositTokenAddress[i + 1],
                    _admins,
                    _onlyAllowWhitelist,
                    _merkleRoot,
                    0,
                    msg.sender
                );
-               ICommLayer(commLayer).sendMsg{value: (msg.value - fees) /
- (_depositChainIds.length - (i + 1))}(
+               uint256 calculatedDeposit = valueToDistribute /
+ (_depositChainIds.length - (i + 1));
+               ICommLayer(commLayer).sendMsg{value: calculatedDeposit}(
                    commLayer, _payload, abi.encode
                      (_depositChainIds[i + 1], msg.sender)
                );
+               valueToDistribute = valueToDistribute - calculatedDeposit;
                unchecked {
                    ++i;
                }
            }
```

[H-04] Reversion in `ApiologyDAOToken` transfer due to insufficient governance tokens

Severity

Impact: High

Likelihood: Medium

Description

The `ApiologyDAOToken` contract allows for the dynamic adjustment of the number of governance tokens associated with each NFT through the `ApiologyDAOToken::changeGovTokensPerNFT` function.

```
File: ApiologyDAOToken.sol
333:     function changeGovTokensPerNFT(uint256 _newAmount) external onlyOwner {
334:         require(_newAmount > 0, "Amount must be greater than 0");
335:>>>      GOVERNANCE_TOKENS_PER_NFT = _newAmount;
336:     }
```

This function directly impacts the `ApiologyDAOToken::_burnGovernanceTokens` function, which is invoked during token transfers from regular addresses to the liquid backing treasury or auction house. If the `GOVERNANCE_TOKENS_PER_NFT` is increased, users may not possess enough governance tokens to cover the required amount for burning (ApiologyDAOToken.sol#L318), leading to a reversion of the transfer transaction.

```
File: ApiologyDAOToken.sol
185:     function transferFrom(
  addressfrom,
  addressto,
  uint256tokenId
) public payable override(ERC721A
...
191:         // Burn governance tokens if transferring from a regular address to
// LBT or Auction House
192:         if (
193:             (to == liquidBackingTreasury || to == auctionHouse)
194:                 && (from != liquidBackingTreasury && from != auctionHouse)
195:         ) {
196:>>>          _burnGovernanceTokens(from, 1);
197:         }
198:
199:         super.transferFrom(from, to, tokenId);
200:
201:         // Mint governance tokens if transferring from LBT or Auction House
// to a regular address
202:         if (
203:             (from == liquidBackingTreasury || from == auctionHouse)
204:                 && (to != liquidBackingTreasury && to != auctionHouse)
205:         ) {
206:             _mintGovernanceTokens(to, 1);
207:         }
...
...
316:     function _burnGovernanceTokens(address from, uint256 amount) internal {
317:         for (uint256 i = 0; i < stationXTokens.length; i++) {
318:>>>          IERC20DAO(stationXTokens[i]).burn
  (from, amount * GOVERNANCE_TOKENS_PER_NFT * 1 ether);
319:         }
320:     }
```

This will cause users to be unable to add tokens to the `auctionQueue` using the `ApiologyAuctionHouse::addToAuctionQueue` function, as the transfer of tokens from the user to the `auctionHouse` contract will be reverted. This will result in

losses for users who wish to sell their tokens, as the auction is the only way to do so. Transferring the NFT to other locations is not allowed, making the auction mechanism the sole avenue for selling.

On the other hand, when the value of `GOVERNANCE_TOKENS_PER_NFT` is reduced, a token transfer from the user to the `auctionHouse` will result in only some governance tokens being burned, leaving the remaining governance tokens with the user. This creates an inconsistency in the burning process, potentially allowing users to retain governance tokens that should have been fully burned during the transfer.

Recommendations

Consider the need to modify the value of `GOVERNANCE_TOKENS_PER_NFT`. If adjustments are necessary, ensure that the `_burnGovernanceTokens` function is updated to burn the correct number of tokens whenever `GOVERNANCE_TOKENS_PER_NFT` is increased or decreased.

Similarly, there should be a mechanism to update a user's governance tokens once `GOVERNANCE_TOKENS_PER_NFT` is changed.

[H-05] Potential overwriting of auction storage leading to loss of auctioned tokens

Severity

Impact: High

Likelihood: Medium

Description

The `ApiologyDAOAuctionHouse` contract utilizes random numbers generated by the Pyth Entropy contract to create auctions for the queued tokens. When a random number is ready to use, the `ApiologyDAOAuctionHouse::entropyCallback` function is invoked to handle the response and initiate the auction creation process.

```
File: ApiologyDAOAuctionHouse.sol
358:     function entropyCallback(
  uint64sequenceNumber,
  address_providerAddress,
  bytes32randomNumber
) internal override {
359:         emit RandomNumberReceived(sequenceNumber, randomNumber);
360:>>       _createAuctionWithRandomNumber(randomNumber);
361:     }
```

The problem arises when the `ApiologyDAOAuctionHouse::entropyCallback` function is called while there is an ongoing auction. This behavior could result in the `auctionStorage` being overwritten, leading to the loss of previously registered auction data.

```
File: ApiologyDAOAuctionHouse.sol
378:     function _createAuctionWithRandomNumber
  (bytes32 randomNumber) internal {
...
414:
415:>>       auctionStorage = AuctionV2({
416:             apdaoId: uint96(apdaoId),
417:             clientId: 0,
418:             amount: 0,
419:             startTime: startTime,
420:             endTime: endTime,
421:             bidder: payable(0),
422:             settled: false
423:         });
...
426:     }
```

Consider the following scenario:

1. The owner calls the pause function and settles to close the current auction.
2. The owner calls the unpause function, triggering a call to `ApiologyDAOAuctionHouse::requestRandomNumber`, which invokes `Entropy::requestWithCallback`.
3. The owner once again calls pause and unpause, triggering another call to `ApiologyDAOAuctionHouse::requestRandomNumber`, which invokes `Entropy::requestWithCallback`.
4. Finally, entropy calls `ApiologyDAOAuctionHouse::entropyCallback` twice, resulting in the auctionStorage being overwritten.

It is important to emphasize that the call to the `ApiologyDAOAuctionHouse::entropyCallback` function occurs in a separate transaction, according to the documentation: *When the final random number is ready to use, the entropyCallback function will be called by the Entropy contract. This will happen in a separate transaction submitted by the requested*

provider On the other hand, we must not assume that `entropy` will not call the callback multiple times.

This results in a situation where the `auctionStorage` is overwritten, causing the token that is currently in the auction to be lost, including the associated bid amount.

Recommendations

Before creating a new auction, check the current state of the `auctionStorage` to ensure that there is no ongoing auction.

```
function entropyCallback(
     uint64sequenceNumber,
     address_providerAddress,
     bytes32randomNumber
   ) internal override {
       emit RandomNumberReceived(sequenceNumber, randomNumber);
-       _createAuctionWithRandomNumber(randomNumber);
+       if (auctionStorage.settled) { _createAuctionWithRandomNumber
+ (randomNumber); };
   }
```

8.3. Medium Findings

[M-01] An auction created when there is no treasury backing

Severity

Impact: High

Likelihood: Low

Description

Whenever an auction is created, we set the reserve price like this:

```
uint256 reservePriceWithBuffer = currentRFV +
  (currentRFV * reservePriceBuffer) / 100;

_setReservePrice(uint192(reservePriceWithBuffer));
```

`currentRFV` is the result of `Treasury::realFloorValue()` where if we have no backing (and no loaned out backing), we will get a value of 0 (or if the float value is higher than the backing). The chance of that happening is slim but the impact is very serious as it can lead to a permanently stuck NFT.

If the reserve price is 0, then that means that someone can bid on an auction using a `msg.value` of 0. This will result in the following 2 variables changes:

```
auctionStorage.amount = uint128(msg.value);
auctionStorage.bidder = payable(msg.sender);
```

The bidder will be the `msg.sender` and the amount will be 0. Now, whenever time passes and the auction gets settled with an amount still equal to 0 (if backing is 0, this is likely to happen as users wouldn't be interested in bidding with actual money), the NFT will be stuck. This is because we will end up in the **else** block in `_settleAuction` which is for when there has been a bidder. There, the whole block is inside this check:

```
if (_auction.amount > 0)
```

As the amount is 0, we wouldn't end up in this check, thus none of the state changes and transfers that are in that block would happen. Now, the NFT is completely stuck as it will stay in the contract. The `nftOwners` mapping will actually still have the user as the NFT owner however upon the auction creation, we pop out that NFT from the auction queue, thus the user can not remove his NFT from the auction queue to get his NFT as it was already removed from there.

Recommendations

Implement the following **else** block when the amount is 0:

```
if (_auction.amount > 0) {
        ...
} else {
     auctionQueue.push(_auction.apdaoId);
}
```

This will put the NFT back into the queue so the user will be able to get it back if the amount was 0 and there was a bidder.

[M-02] `mintToken()` assumes that the user we mint to is not an admin

Severity

Impact: Low

Likelihood: Medium

Description

Upon `ERC20DAO::mintToken()` is called, we have this piece of code:

```
Emitter(emitterContractAddress).newUser(
            address(this),
            to,
            Factory(factoryAddress).getDAOdetails(address
              (this)).depositTokenAddress,
            0,
            block.timestamp,
            amount,
            false
        );
```

The last variable is whether the user is an admin. As seen, it always assumes that the user is not an admin. However, that is an incorrect assumption as there are multiple scenarios where the user might be an admin. Firstly, we have this line when transferring an NFT:

```
if ((from == liquidBackingTreasury || from == auctionHouse) &&
          (to != liquidBackingTreasury && to != auctionHouse)) {
            _mintGovernanceTokens(to, 1);
        }
```

If the `from` is the auction house and the `to` is not, we will mint a governance token which calls the function discussed initially. This scenario happens when someone removes his NFT from the auction queue which is a completely expected thing to do by an admin. This function also gets called upon someone calling `ApiologyDAOToken::claim()` which is again something that can be called by an admin.

Recommendations

Change the `false` to the code being used in `ERC20DAO::mintGTToAddress()`.

[M-03] `setPrices()` accesses the settlement history by a wrong value

Severity

Impact: Medium

Likelihood: Medium

Description

Let's take a look at the `setPrices()` function:

```
function setPrices
  (SettlementNoClientId[] memory settlements) external onlyOwner {
        for (uint256 i = 0; i < settlements.length; ++i) {

                        SettlementState storage settlementState = settlementHistory[s
            settlementState.blockTimestamp = settlements[i].blockTimestamp;
            settlementState.amount = ethPriceToUint64(settlements[i].amount);
            settlementState.winner = settlements[i].winner;
            settlementState.apdaoId = uint96(settlements[i].apdaoId);
        }
    }
```

As seen, we access the settlement history by the NFT ID. However, that is not in sync with every other place in the code. For example, let's take a look at how the settlement history is set when an auction is settled:

```
uint256 settlementId = currentSettlementId++;
SettlementState storage settlementState = settlementHistory[settlementId];
```

We access the settlement history linearly based on the current settlement ID. However, in the above case, we access it by the NFT ID which doesn't follow such a linearity and these 2 values are completely unrelated. Some functions also assume that settlements are in order which might be wrong due to the way `setPrices()` works.

Recommendations

Include a second input, an array of settlement IDs, and use them to access the settlement history instead. Also, include a length check for the 2 arrays for sanity purposes.

[M-04] Auction creation during paused state due to entropy callback

Severity

Impact: Medium

Likelihood: Medium

Description

The `ApiologyAuctionHouse` contract has a mechanism to pause its operations preventing new auctions from being created. The creation of an auction can

only occur through the `ApiologyAuctionHouse::unpause` and `ApiologyAuctionHouse::settleCurrentAndCreateNewAuction` functions, and these two functions can only be executed if the contract is not paused. However, the `entropyCallback` function can create a new auction without checking if the contract is paused. This occurs because the entropy callback can be triggered asynchronously after a request for a random number is made, potentially during a time when the contract has since been paused. This behavior allows the creation of auctions even when the contract is intended to be inactive, violating the contract's pause state control.

```
File: ApiologyDAOAuctionHouse.sol
358:     function entropyCallback(
  uint64sequenceNumber,
  address_providerAddress,
  bytes32randomNumber
) internal override {
359:         emit RandomNumberReceived(sequenceNumber, randomNumber);
360:         _createAuctionWithRandomNumber(randomNumber);
361:     }
```

This function is called by the entropy provider in response to a random number request, and it directly calls `_createAuctionWithRandomNumber`, potentially creating a new auction without checking the pause state.

Consider the following scenario:

1. The contract is about to be paused for various reasons, and before that happens, a request for a random number is made. After the request, the contract is paused.
2. In a different transaction, `entropy` calls `entropyCallback`, creating a new auction even though the contract is paused.

Recommendations

Implement a check within the `ApiologyAuctionHouse::entropyCallback` function to verify if the contract is paused before proceeding with auction creation.

[M-05] Failure of entropy source callback

Severity

Impact: High

Likelihood: Low

Description

The `ApiologyAuctionHouse` contract relies on an external entropy source, implemented through the `Pyth Entropy` contract, to provide random numbers necessary for creating auctions. The main function responsible for handling the random number response is `ApiologyAuctionHouse::entropyCallback`, which is supposed to be invoked by the entropy source. If this callback function is not called, the contract cannot proceed with creating new auctions that require randomness.

```
File: ApiologyDAOAuctionHouse.sol
358:     function entropyCallback(
  uint64sequenceNumber,
  address_providerAddress,
  bytes32randomNumber
) internal override {
359:         emit RandomNumberReceived(sequenceNumber, randomNumber);
360:>>>      _createAuctionWithRandomNumber(randomNumber);
361:     }
```

Without the callback, the auction creation process halts, preventing new auctions from being initiated. The inability to proceed automatically necessitates manual intervention, where the owner must pause and unpause the contract to reset auction creation mechanisms which introduces a central point of failure and potential misuse if the owner is unavailable or acts maliciously.

Recommendations

Implement a fallback mechanism to re-request a random number if the callback is not received within a certain timeframe. This can be done using a timeout or a retry counter.

[M-06] Unhandled exceptions in a function

Severity

Impact: Medium

Likelihood: Medium

Description

In the `ApiologyAuctionHouse::_createAuctionWithRandomNumber` function, there is a try/catch block that only catches `Error(string memory)`.

```
File: ApiologyDAOAuctionHouse.sol
402:         } else {
403:             try apiologyToken.mint() returns (uint256 mintedapDAOId) {
404:                 apdaoId = mintedapDAOId;
405:                 nftOwners[apdaoId] = address(0);
406:             } catch Error(string memory) {
407:                 _pause();
408:                 return;
409:             }
410:         }
```

This approach is insufficient because the `apiologyToken::mint` function can be paused, triggering a `revert EnforcedPause();` which is not captured by the current try/catch implementation.

> catch Error(string memory reason) { ... }: This catch clause is executed if the error was caused by revert("reasonString") or require(false, "reasonString") (or an internal error that causes such an exception).

As a result, if the token minting is paused, the auction creation process fails, and a new random number request is initiated without verifying the paused state. This oversight leads to unnecessary entropy fee expenditure by the user, as the random number request incurs a cost each time it is called.

Recommendations

Enhance the try/catch block to handle exceptions related to the paused state.

```
} else {
            try apiologyToken.mint() returns (uint256 mintedapDAOId) {
                apdaoId = mintedapDAOId;
                nftOwners[apdaoId] = address(0);
-           } catch Error(string memory) {
+           } catch {
                _pause();
                return;
            }
        }
```

[M-07] `createERC20DAO()` will always revert when `_depositChainIds` has a length bigger

than 1

Severity

Impact: Medium

Likelihood: Medium

Description

Whenever we call `createERC20DAO()`, we have the following **if** check:

```
if (_depositChainIds.length > 1)
```

If we end up in there, we will always revert. This is due to this line:

```
IEmitter(emitterAddress).createCCDao(_daoAddress, _depositChainIds);
```

It calls `createCCDao()` on the `Emitter` contract however that function is not implemented there. This can easily be confirmed by searching for that function name by using `CTRL + Shift + F` and we will see that the only function with that name across all files is located in the `IEmitter` interface.

Recommendations

Implement the `createCCDao()` functionality

[M-08] Bias in random token selection due to modulo operator

Severity

Impact: Medium

Likelihood: Medium

Description

The function `_createAuctionWithRandomNumber` uses a modulo operation to select a random index from the `auctionQueue`:

```
File: ApiologyDAOAuctionHouse.sol
378:     function _createAuctionWithRandomNumber
  (bytes32 randomNumber) internal {
...
390:         if (auctionQueue.length > 0) {
391:>>>          uint256 randomIndex = uint256
  (randomNumber) % auctionQueue.length;
392:             apdaoId = auctionQueue[randomIndex];
393:
394:             // Remove the selected token from the queue
395:
               auctionQueue[randomIndex] = auctionQueue[auctionQueue.length - 1];
396:             auctionQueue.pop();
397:
398:             // Emit the removal event
399:             uint256[] memory removedTokens = new uint256[](1);
400:             removedTokens[0] = apdaoId;
401:             emit TokensRemovedFromAuctionQueue
  (removedTokens, nftOwners[apdaoId]);
402:         } else {
...
```

This can lead to an uneven distribution of selection probabilities across the tokens in the queue, the pyth documentation says:

> Notice that using the modulo operator can distort the distribution of random numbers if it's not a power of 2. This is negligible for small and medium ranges, but it can be noticeable for large ranges. For example, if you want to generate a random number between 1 and 52, the probability of having value 5 is approximately 10^-77 higher than the probability of having value 50 which is infinitesimal.

In this case, if `auctionQueue.length` does not evenly divide the range of `randomNumber`, some indices may have a higher probability of being chosen than others. This can lead to an uneven selection process, potentially favoring certain NFTs over others in the auction queue. This is particularly concerning in large queues.

Recommendations

To mitigate the bias introduced by the modulo operation, consider using a different method to select a random index.

8.4. Low Findings

[L-01] Mismatch between checks for storage variables

In the `initialize()` function in the auction house, we set the following variables:

```
duration = _duration;
reservePrice = _reservePrice;
timeBuffer = _timeBuffer;
minBidIncrementPercentage = _minBidIncrementPercentage;
feePercentage = 0;
```

For every single one of those variables, there is a setter function and for every single one of those variables except the reserve price, we have a sanity check, for example:

```
function setAuctionDuration(uint256 _duration) external onlyOwner {
        require(_duration > 0, "Duration must be greater than zero");
        duration = _duration;
        emit AuctionDurationUpdated(_duration);
    }
```

As seen, the `initialize()` function does not have any of these checks, there is a mismatch. If you are confident to not set any wrong values, you can remove the checks in the setter functions to optimize the contract. However, if you want to keep your contract as safe as possible, consider implementing those sanity checks in the initialize function as well.

[L-02] Difference between emitted events

`ERC20DAO::mintToken()` and `ERC20DAO::mintGTToAddress()` are both very identical in terms of their purpose. The first one mints governance tokens to a specified address while the latter mints governance tokens to multiple addresses. Thus, these functions are expected to emit the same events and possibly, one of them might emit an extra event(s) to show this was a batch mint or some other extra information. However, the latter one completely misses this event emission:

```
Emitter(emitterContractAddress).deposited(...);
```

Instead, consider adding it as well.

[L-03] Not checking of being done upon the ERC20 DAO creation

Upon creating an ERC20 DAO, we have this check:

```
if ((
          (_distributionAmount * _pricePerToken) / 1e18) < _maxDepositPerUser) {
            revert RaiseAmountInvalid(((

            ) / 1e18
        }
```

Then, in `updateTotalRaiseAmount()` which aims to update the distribution amount and price per token, we first have this check:

```
if (_distributionAmount != _newDistributionAmount) {
            if (_distributionAmount > _newDistributionAmount) {
                revert AmountInvalid
                  ("_newDistributionAmount", _newDistributionAmount);
            }
         ...
        }
```

We can see that if the distribution amount is updated to a lower value, we will revert. This implicitly makes sure that the initial check shown in the beginning would still pass as if the distribution amount is going up, then there is no way we go below the `_maxDepositPerUser` variable if we were above it beforehand. Then, we have this code:

```
if (daoDetails[_daoAddress].pricePerToken != _newPricePerToken) {
            daoDetails[_daoAddress].pricePerToken = _newPricePerToken;
            IEmitter(emitterAddress).updatePricePerToken
              (_daoAddress, _newPricePerToken);
        }
```

We can see that there is no check regarding the price per token here which allows the initial check to not be valid anymore as a price per token of 0, for example, would always cause the initial check to not be valid anymore.

[L-04] `updateDepositTime()` allows for setting the deposit close time in the past

`updateDepositTime()` has the following input:

> /// @param _depositTime New deposit close time

We can see that the input is the deposit close time. Upon calling that function, we only have this check regarding that input:

```
if (_depositTime == 0) revert AmountInvalid("_days", _depositTime);
```

Thus, the deposit close time can actually be set in the past as it is not validated against `block.timestamp`. Instead, change the above check to the following:

```
if (_depositTime <= block.timestamp) revert AmountInvalid
  ("_days", _depositTime);
```

[L-05] Mismatch between the minimum and maximum deposits

Whenever we create an ERC20 DAO, we have this line for the minimum and maximum user deposit:

```
if (_maxDepositPerUser <= _minDepositPerUser) {
            revert DepositAmountInvalid(_maxDepositPerUser, _minDepositPerUser);
        }
```

Let's say that both the maximum and minimum deposit are 10000. As they are equal, this would revert. However, we also have this function for updating them:

```
function updateMinMaxDeposit(
      uint256_minDepositPerUser,
      uint256_maxDepositPerUser,
      address_daoAddress
    )
        external
        payable
        onlyGnosisOrDao(address(this), _daoAddress)
    {
        ...

        validateDepositAmounts(_minDepositPerUser, _maxDepositPerUser);
        ...
    }
```

If we take a look at the function responsible for validating, we see this:

```
if (_min == 0 || _min > _max) revert DepositAmountInvalid(_min, _max);
```

Let's use those same values we used above. Since 10000 is not bigger than 10000, this will not revert. As seen, these 2 functions validate differently. Instead, decide which way is your intended one and stick to it in both functions.

[L-06] Multiple instances of wrongly assuming a settlement state is empty

In a few functions such as `ApiologyDAOAuctionHouse::getSettlements()`, we have code similar to this:

```
settlementState = settlementHistory[id];

            if (skipEmptyValues && settlementState.winner == address
              (0)) continue;

            settlements[actualCount] = Settlement({
                blockTimestamp: settlementState.blockTimestamp,
                amount: uint64PriceToUint256(settlementState.amount),
                winner: settlementState.winner,
                apdaoId: settlementState.apdaoId,
                clientId: settlementState.clientId,
                settlementId: id
            });
```

Let's focus on this line:

```
if (skipEmptyValues && settlementState.winner == address(0)) continue;
```

The goal of this line is to skip settlements that are empty. However, an `address(0)` winner is not exactly an empty settlement. The winner can be `address(0)` when there have been no bidders for an auction. However, that is still written into the `settlementState` storage:

```
SettlementState storage settlementState = settlementHistory[settlementId];
        settlementState.blockTimestamp = uint32(block.timestamp);
        settlementState.amount = ethPriceToUint64(_auction.amount);
        settlementState.winner = _auction.bidder;
        settlementState.apdaoId = _auction.apdaoId;
```

The values that are changed above are the timestamp and the NFT ID which are values that could be of use for users calling a function such as `ApiologyDAOAuctionHouse::getSettlements()` but they will not be able to as they are assumed to be empty.

[L-07] The `factory.emitSignerChanged` function allows any `dao` address

```
/// @file: contracts/stationX/factory.sol
    function emitSignerChanged(address _dao, address _signer, bool _isAdded)
        external
        onlyGnosisOrDao(address(this), _dao)
    {
        IEmitter(emitterAddress).changedSigners(_dao, _signer, _isAdded);
    }
```

The caller can pass in the address of the `_dao` deployed by itself to bypass the `onlyGnosisOrDao(address(this), _dao)` modifier to emit a `changedSigner` event.

[L-08] The `factory.createERC20DAO` function may deploy an empty Gnosis Safe contract

The `factory.createERC20DAO` function will call `IDeployer(deployer).deploySAFE` to deploy the Safe contract. The `deploySAFE` function code is shown below.

```
/// @file: contracts/stationX/Deployer.sol
    function deploySAFE
      (address[] calldata _admins, uint256 _safeThreshold, address _daoAddress)
        external
        returns (address SAFE)
    {
        bytes memory _initializer = abi.encodeWithSignature(
            "setup(
              address[],
              uint256,
              address,
              bytes,
              address,
              address,
              uint256,
              address
            )",
            _admins,
            _safeThreshold,
            0x0000000000000000000000000000000000000000,
            "",
            safeFallback,
            0x0000000000000000000000000000000000000000,
            0,
            0x0000000000000000000000000000000000000000
        );

        uint256 nonce = getNonce(_daoAddress);
A>      try ISafe(safe).createProxyWithNonce
  (singleton, _initializer, nonce) returns (address _deployedSafe) {
            SAFE = _deployedSafe;
        } catch {}
    }
```

As shown in code A, the `deploySAFE` function allows the deployment of the Safe contract to fail. If it fails, an empty address will be returned.

[L-09] The factory does not check whether the create fees are paid successfully

The `factory.createERC20DAO` function requires the caller to pay create fees. The code is as follows.

```
/// @file: contracts/stationX/factory.sol
    function checkCreateFeesSent
      (uint256[] calldata _depositChainIds) internal returns (uint256) {
        require(_depositChainIds.length > 0, "Deposit Chains not provided");
        uint256 fees = createFees * _depositChainIds.length;
        require(msg.value >= fees, "Insufficient fees");
A>      payable(_owner).call{value: fees}("");
        return fees;
    }
```

However, in code A, it does not check whether the `call` is successful. If the `call` fails, the `_owner` will not receive the create fees.

[L-10] Losing funds because the contract forces the use of WETH when ETH transfer fails

The `ApiologyAuctionHouse` contract usually uses ETH as the payment token. However, when the ETH transfer fails, it will force the use of WETH token transfer. The code is as follows.

```
/// @file: contracts/ApiologyDAOAuctionHouse.sol
    function _safeTransferETHWithFallback(address to, uint256 amount) internal {
        if (!_safeTransferETH(to, amount)) {
            IWETH(weth).deposit{value: amount}();
            IERC20(weth).transfer(to, amount);
        }
    }
```

This means that if the user has a contract that does not support ERC20, his funds will be lost.

[L-11] Absence of event emissions in critical functions

Several critical functions do not emit events upon execution. These functions include `ApiologyDAOAuctionHouse::setPrices`, `ApiologyDAOAuctionHouse::setEntropy`, `erc20dao::updateGovernanceActive`, `Deployer::transferOwnerShip`, `Deployer::defineContracts`, and `Factory::defineTokenContracts`. The absence of event emissions can lead to reduced transparency and traceability of important state changes within the smart contract ecosystem.

Implement event emissions for the identified functions. Each function should emit an event that logs the relevant changes and data involved.

[L-12] Redundant role assignment

In the `createDaoErc20` function within the `Emitter` contract, there is a line that grants the `EMITTER` role to `msg.sender`:

```
File: emitter.sol
161:     function createDaoErc20(
162:         address _deployerAddress,
163:         address _proxy,
164:         string memory _name,
165:         string memory _symbol,
166:         uint256 _distributionAmount,
167:         uint256 _pricePerToken,
168:         uint256 _minDeposit,
169:         uint256 _maxDeposit,
170:         uint256 _ownerFee,
171:         uint256 _totalDays,
172:         uint256 _quorum,
173:         uint256 _threshold,
174:         address _depositTokenAddress,
175:         address _emitter,
176:         address _gnosisAddress,
177:         address lzImpl,
178:         bool _isGovernanceActive,
179:         bool isTransferable,
180:         bool assetsStoredOnGnosis
181:     ) external payable onlyRole(FACTORY) {
182:         _grantRole(EMITTER, _proxy);
183:>>>      _grantRole(EMITTER, msg.sender);
```

Given that the function is restricted by the `onlyRole(FACTORY)` modifier, `msg.sender` will always be the `FACTORY` contract. Since the `FACTORY` contract already has the `EMITTER` role, this operation is redundant. Continually executing this line on every call to `createDaoErc20` results in unnecessary gas consumption.

Eliminate the line that grants the `EMITTER` role to `msg.sender` within the `createDaoErc20` function. This will reduce unnecessary gas usage and simplify the role management logic.

[L-13] Unnecessary use of `payable` modifier

The functions `defineContracts` and `createDaoErc20` in the `emitter.sol` contract are marked with the `payable` modifier. However, they do not handle `msg.value`, which implies that they are not intended to receive Ether transactions. The presence of the `payable` modifier in these functions is unnecessary and could lead to confusion or misinterpretation of the contract's capabilities.

```
File: emitter.sol
151:     function defineContracts(
152:         address ERC20ImplementationAddress,
153:         address ERC721ImplementationAddress,
154:         address emitterImplementationAddress
155:>>   ) external payable onlyRole(FACTORY) {
...
...
161:     function createDaoErc20(
...
181:>>   ) external payable onlyRole(FACTORY) {
```

Update the function definitions to remove the `payable` modifier, ensuring that only functions intended to receive Ether can do so, reducing potential confusion.

[L-14] Unnecessary random number request with single NFT in queue

The function `ApiologyAuctionsHouse::_createAuction` currently makes a request for a random number even when there is only one NFT in the `auctionQueue`. This is inefficient as there is no need for random selection when only one item is available. This results in additional, unnecessary costs for users due to fees associated with the random number request.

```
File: ApiologyDAOAuctionHouse.sol
366:     function _createAuction() internal {
367:         if (auctionQueue.length > 0) {
368:             requestRandomNumber();
369:         } else {
370:             _createAuctionWithRandomNumber(0);
371:         }
372:     }
```

Introduce a conditional check to determine if there is only one NFT in the `auctionQueue`. If so, bypass the random number request and proceed directly to auction creation.

```
function _createAuction() internal {
-       if (auctionQueue.length > 0) {
+       if (auctionQueue.length > 1) {
            requestRandomNumber();
        } else {
            _createAuctionWithRandomNumber(0);
        }
    }
```

[L-15] Redundant gnosis safe deployment in `factory::createERC20DAO`

In the `factory::createERC20DAO` function, there is a conditional check that determines whether a new Gnosis Safe should be deployed. Specifically, if the `_depositChainIds` array contains more than one element, the function overrides any existing Gnosis Safe address provided by the user and proceeds to deploy a new one. This logic can lead to the unnecessary deployment of a new Gnosis Safe, even if the user has an existing one they intend to use.

```
File: factory.sol
185:         if (_gnosisAddress == address(0) || _depositChainIds.length > 1) {
186:>>>          _safe = IDeployer(deployer).deploySAFE
  (_admins, _safeThreshold, _daoAddress);
187:         } else {
188:             if (!isGnosisSafe(_gnosisAddress)) {
189:                 revert InvalidGnosisSafe
  ("Invalid Gnosis Safe", _gnosisAddress);
190:             }
191:
192:             _safe = _gnosisAddress;
193:         }
```

It is recommended to modify the logic to respect the user's provided Gnosis Safe address regardless of the length of `_depositChainIds`.

[L-16] Insufficient validation of `maxDepositPerUser` and `pricePerToken`

In the `updateMinMaxDeposit` function, there is a lack of validation to ensure that `maxDepositPerUser` does not exceed the calculable deposit limit derived from `(_distributionAmount * _pricePerToken) / 1e18`:

```
File: factory.sol
355:     function updateMinMaxDeposit
  (uint256 _minDepositPerUser, uint256 _maxDepositPerUser, address _daoAddress)
356:         external
357:         payable
358:         onlyGnosisOrDao(address(this), _daoAddress)
359:     {
360:         validateDaoAddress(_daoAddress);
361:
362:         validateDepositAmounts(_minDepositPerUser, _maxDepositPerUser);
363:
364:         daoDetails[_daoAddress].minDepositPerUser = _minDepositPerUser;
365:         daoDetails[_daoAddress].maxDepositPerUser = _maxDepositPerUser;
366:
367:         IEmitter(emitterAddress).updateMinMaxDeposit
  (_daoAddress, _minDepositPerUser, _maxDepositPerUser);
368:     }
```

This validation is performed when the `erc20dao` is initially created; however, when `maxDeposit` is updated, the validation is no longer executed.

```
File: factory.sol
331:         if ((
  (_distributionAmount * _pricePerToken) / 1e18) < _maxDepositPerUser) {
332:             revert RaiseAmountInvalid((
  (_distributionAmount * _pricePerToken) / 1e18), _maxDepositPerUser);
333:         }
```

Add a validation check in the `factory::updateMinMaxDeposit` function to ensure that the `maxDepositPerUser` does not exceed the calculated distribution value `((_distributionAmount * _pricePerToken) / 1e18)`.

[L-17] Absence of the payable modifier for request fees

The `ApiologyDAOAuctionHouse::requestRandomNumber` function sends ETH as a `requestFee` when calling the `entropy::requestWithCallback` function.

```
File: ApiologyDAOAuctionHouse.sol
344:     function requestRandomNumber() internal {
345:         bytes32 userRandomNumber = keccak256(abi.encodePacked
  (block.timestamp, msg.sender));
346:         address provider = entropy.getDefaultProvider();
347:         uint128 requestFee = entropy.getFee(provider);

349:         emit RandomNumberRequested(sequenceNumber);
350:     }
```

This function is invoked from `ApiologyDAOAuctionHouse::_createAuction`, which can be triggered by `ApiologyDAOAuctionHouse::unpause()` and `ApiologyDAOAuctionHouse::settleCurrentAndCreateNewAuction()` methods. Neither of these methods has the `payable` modifier, indicating that the contract may not have enough ETH to cover the fee unless it has a pre-existing balance.

Recommendations

Consider adding the `payable` modifier to the `unpause()` and `settleCurrentAndCreateNewAuction()` functions, and ensure that the user sends the correct `requestFee` amount if a random number from `Entropy` is required. This will ensure that the necessary funds are provided for the randomness number request.

[L-18] `createERC20DAO()` uses the deposit time input in a wrong way

`createERC20DAO()` has the following input:

> /// @param _depositTime Deposit period duration

As seen, the input is the deposit period duration. However, if we take a look at its usage:

```
daoDetails[_daoAddress] = DAODetails(
            _pricePerToken,
            _distributionAmount,
            _minDepositPerUser,
            _maxDepositPerUser,
            _ownerFeePerDepositPercent,
            _depositTime, <@
            _depositTokenAddress,
            _gnosisAddress,
            _merkleRoot,
            true,
            false,
            _assetsStoredOnGnosis
        );
```

Based on the struct, is this variable:

```
uint256 depositCloseTime;
```

It confuses the duration as the close time. Thus, a duration of 100 minutes would set the close time equal to 100 minutes in seconds which is very far in the past.

If the comment is correct, then add that variable to `block.timestamp`. If the comment is wrong, change it.

[L-19] Malicious bidders can disincentivize users from bidding

Severity

Impact: Medium

Likelihood: Medium

Description

Upon someone bidding, if there is a previous bidder, we refund him the amount that he bid using the function below:

```
function _safeTransferETHWithFallback(address to, uint256 amount) internal {
        if (!_safeTransferETH(to, amount)) {
            IWETH(weth).deposit{value: amount}();
            IERC20(weth).transfer(to, amount);
        }
    }
```

We first call `_safeTransferETH()` and if that fails, we will convert that amount of ETH into WETH and transfer it to the user. This is done in order to disallow bidders from blocking the transfer by reverting to their `receive()` function. Let's take a look at `_safeTransferETH()`:

```
function _safeTransferETH(address to, uint256 value) internal returns
      (bool) {
        bool success;
        assembly {
            success := call(70000, to, value, 0, 0, 0, 0)
        }
        return success;
    }
```

We use assembly to transfer the funds to the user by limiting the gas to 70000. This disallows the receiver from using up all of the gas which would make the

new bidder's transaction revert and disallow users from bidding. However, the user can still use up the 70000 gas and make the function revert. The revert will be caught and WETH will be transferred to him which won't block the function however if we take a look at the average gas price from a recent day, we will see 27 gwei. To calculate the cost of that, we multiply $27 * 70000$ resulting in 1890000 gwei which at current prices is around 4.50$. Thus, absolutely every bidder can increase the costs of a bid by 4.50$ which is not a small amount of funds (note that this is just the increase caused by the malicious bidder, not the total gas costs). Combining that with the fact that the gas prices can be much higher than that (just in the last year, the highest average gas price for a day was 100 making the increase 4 times bigger) and the price of ETH can go up significantly, this is a concern - users will be charged more for fees and can be disincentivized from bidding as that increase could make their bid not worth it for them.

Recommendations

Limiting the gas further is an option however that could make failures more common. If you want to keep the push over pull pattern, I would recommend not sending out ETH at all but directly converting the amounts to WETH and transferring that instead, this will make such a scenario impossible as well as any DoS vectors by a bidder while keeping the great UX of the push over pull pattern.

[L-20] Attackers can frontrun and increase refund gas to DoS the bid

When UserA submits a new bid, the funds from the previous bid will be returned to the previous bidder (assuming it is UserB). The problem is that UserB may be able to DoS UserA's bid by frontrunning raising the refund gas. It relies on the refund mechanism, the code is as follows.

```
/// @file: contracts/ApiologyDAOAuctionHouse.sol
    function _safeTransferETHWithFallback(address to, uint256 amount) internal {
        if (!_safeTransferETH(to, amount)) {
            IWETH(weth).deposit{value: amount}();
            IERC20(weth).transfer(to, amount);
        }
    }
--snip--
    function _safeTransferETH(address to, uint256 value) internal returns
      (bool) {
        bool success;
        assembly {
            success := call(70000, to, value, 0, 0, 0, 0)
        }
        return success;
    }
```

The `_safeTransferETHWithFallback` will `call(70000, to, value, 0, 0, 0, 0)` to transfer the ETH. The `to` is a contract that is fully controllable by UserB. Then the attack steps are as follows.

1. Before UserA calls `createBid`, he usually simulates the execution and obtains the gas consumption. We assume that it is 100,000
2. UserA increases the gas by 20% for redundancy, so when he executes `createBid`, he gives a gaslimit of 120,000
3. UserB frontrunning increases gas consumption of `to.receive`
4. UserA's tx actually requires 170,000 when executed, so the execution fails.

It is recommended to optimize the refund mechanism, that is, to store the refund in the contract and provide a method for users to claim it.