# Aori

Smart Contract Security Assessment

February 10, 2023

# ABSTRACT

Dedaub was commissioned to perform a security audit of the Aori Protocol. The protocol implements fully-collateralized, decentralized options, and a marketplace for initial agreement of the option contract between the two parties.

This audit report covers the at-this-time private repository https://github.com/elstongun/Aori, at commit hash `62bd59a3a917295b1ebd8a5d8cb5bd7c8727cb42`. Review of fixes was performed at commit `44ddd2ebbd16d563c3494c878134eeb21ea04f89`, over the code changes to items identified in the audit report (i.e., without a complete re-audit on `44ddd2eb`).

## SETTING & CAVEATS

Two auditors worked on the codebase for 5 days on the following contracts:

```
src/
├─ AoriAuctionHouse.sol
├─ AoriCall.sol
├─ AoriPut.sol
├─ AoriSeats.sol
├─ Ask.sol
├─ Bid.sol
├─ Optiontroller.sol
├─ Orderbook.sol
├─ OrderbookLens.sol
│
└─ Interfaces/
   └─ IAoriAuctionHouse.sol
```

In the post-audit revised code, the `Optiontroller` functionality changed into
`CallFactory.sol`
`PutFactory.sol`
`OrderbookFactory.sol`

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Although a high-level specification describing interactions within the protocol was provided, functional correctness (i.e. issues in "regular use") was a secondary consideration. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

The project's test suite has not been fully developed at the time of auditing. We are concerned and recommend very thorough testing of both common usage and corner cases, since most of the issues uncovered during auditing are functional correctness issues and not flaws of the fundamental security model. The trust/security model of the contracts is solid and thorough, considering both standard attack vectors (e.g., reentrancy) and a transitive establishment of trust for callees of sensitive functionality (e.g., trusting the Optiontroller contract at creation, and consulting it as to what other callees can be trusted).

## VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

| Category | Description |
|---|---|
| CRITICAL | Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result. |
| HIGH | Third-party attackers or faulty functionality may block the system or |

| | |
|---|---|
| | cause the system or users to lose funds. Important system invariants can be violated. |
| MEDIUM | Examples:<br>• User or system funds can be lost when third-party systems misbehave.<br>• DoS, under specific conditions.<br>• Part of the functionality becomes unusable due to a programming error. |
| LOW | Examples:<br>• Breaking important system invariants but without apparent consequences.<br>• Buggy functionality for trusted users where a workaround exists.<br>• Security issues which may manifest when the system evolves. |

Issue resolution includes “dismissed” or “acknowledged” but no action taken, by the client, or “resolved”, per the auditors.

## CRITICAL SEVERITY:

| ID | Description | STATUS |
|---|---|---|
| C1 | Misuse of `decimals()` can lead to loss of funds | **RESOLVED (commit ccbf56c)** |
| `AoriCall::sellerSettlementITM` wrongly uses the number of decimals of the underlying token when representing the value of tokens to be received. | | |

```
function sellerSettlementITM(uint256 optionsToSettle) public nonReentrant returns
(uint256) {
…
       require(settlementPrice > strikeInUSDC && hasEnded == true);

       uint256 UNDERLYINGToReceive = ((strikeInUSDC * USDC.decimals()) /
settlementPrice) * optionsSold; // (1e6*1e6/1e6) * 1e18
       //store the settlement
       uint256 newOptionsSold = optionsSold - optionsToSettle;
       optionSellers[msg.sender] = newOptionsSold;

       //settle

       UNDERLYING.transfer(msg.sender, UNDERLYINGToReceive / 10**USDC.decimals());
…
}
```

The above snippet tries to avoid making `UNDERLYINGToReceive` zero by first multiplying `strikeInUSDC` with `USDC.decimals()`. However, `strikeInUSDC` will be multiplied with a small constant (6) and the units of `strikeInUSDC` will not effectively change. The value of `strikeInUSDC` should be multiplied with `10**USDC.decimals()`.

In the general case, because the division of `UNDERLYINGToReceive` with `10**USDC.decimals()` has not been matched with a corresponding multiplication when calculating its value, sellers are going to receive fewer underlying tokens than normal.

Analogous cases of `decimals()` misuse can be found in `AoriCall::buyerSettlementITM`

```
function buyerSettlementITM(uint256 optionsToSettle) public nonReentrant returns
(uint256) {
…
       uint256 profitPerOption = ((settlementPrice - strikeInUSDC) *
USDC.decimals()) / settlementPrice;
       uint256 UNDERLYINGOwed = (profitPerOption * optionsToSettle) /
USDC.decimals(); //1e6 * 1e18 / 1e6
```

```
        _burn(msg.sender, optionsToSettle);
        UNDERLYING.transfer(msg.sender, UNDERLYINGOwed); //sending 1e18 scale
tokens to user
…
}
```

As well as in `Ask::fill`

```
function fill(uint256 amountOfUSDC, uint256 seatId) public nonReentrant {
        …

        if(msg.sender == AORISEATSADD.ownerOf(seatId)) {
                …
                OPTIONToReceive = mulDiv(USDCAfterFee, OPTIONDecimals,
USDCPerOPTION); //1eY = (1eX * 1eY) / 1eX
                …
                //transfer to the Msg.sender
                OPTION.transfer(msg.sender, OPTIONToReceive);
        } else {
                …
                //And the amount of the quote currency the msg.sender will receive
                OPTIONToReceive = mulDiv(USDCAfterFee, OPTIONDecimals,
USDCPerOPTION); //(1e6 * 1e18) / 1e6 = 1e18
                …
                //Transfers to the msg.sender
                OPTION.transfer(msg.sender, OPTIONToReceive);
}
```

| ID | Description | STATUS |
|---|---|---|
| C2 | The settlement price of call options has the wrong units, making the outcome of settlements largely predictable | **RESOLVED (commit 4aa163c)** |

When querying the price of the underlying token from the selected Chainlink aggregator in `AoriCall::getPrice`, the contract attempts to convert the answer so that it is in USDC decimals:

```
function getPrice() public view returns (uint256) {
      (, int256 price, , , ) = AggregatorV3Interface(oracle).latestRoundData();
      return (uint256(price) / (10**8 - 10**USDC.decimals()));
}
```

However, the subtraction In the denominator should be in the exponents of the arithmetic expression.

Consequently, the price that gets returned by `getPrice()` will be considerably smaller than normal (`10**2 < 10**8 - 10**6`). Call options with a strike price that would be comparable to the settlement price (i.e., not orders of magnitude different) could end up being settled out-of-the-money, thus profiting the side of the seller.

## HIGH SEVERITY:

| ID | Description | STATUS |
|---|---|---|
| H1 | `AoriSeats::separateSeats` allows anyone with a seat to reach `maxSeatScore` for an arbitrary number of seats | **RESOLVED (commit ccbf56c)** |

This function mints as many 1-score seats as the current seat score of `seatId`

```
function separateSeats(uint256 seatId) public {
      require(msg.sender == ownerOf(seatId));
      uint256 currentSeatScore = seatScore[seatId];
      for(uint i = 0; i < currentSeatScore; i++) {
            uint mintIndex = totalSupply();
            _safeMint(msg.sender, mintIndex);
            seatScore[mintIndex] = 1;
      }
}
```

However, `AoriSeats::separateSeats` does not burn the `seatId` that gets separated, allowing someone to call the function for the same `seatId` multiple times. For instance, a seat holder can get infinitely many 1-score seats and combine them using `AoriSeats::combineSeats` to reach `maxSeatScore`. The user exploiting this will be able to receive the maximum amount of fee rewards when one of their `seatIds` gets used within the protocol.

| H2 | `AoriPut/AoriCall::setSettlementPrice()` can be called multiple times, with counter-intuitive results | **RESOLVED (commit 0b6dd23)** |
|---|---|---|

The function `setSettlementPrice` ensures neither that it is called atomically with the first settlement nor that it cannot be called again.

```
function setSettlementPrice() public returns (uint256) {
      require(block.number >= endingBlock);
      settlementPrice = uint256(getPrice());
      hasEnded = true;
      return settlementPrice;
}
```

As a result, a buyer or seller of an option can wait for an opportune moment to call the function. Indeed, some amount of the same option can be settled in-the-money with some other being settled out-of-the-money.

Ideally, the settlement price for the entire option should be set once and for all by the first party that settles (as early as possible after the ending block, which is loosely ensured by at least one of the parties having a financial incentive to settle at the current price).

## MEDIUM SEVERITY:

| ID | Description | STATUS |
|---|---|---|
| M1 | AoriSeats::combineSeats can reuse `seatIds` with surprising results | **RESOLVED (commit d343c42)** |

Function `combineSeats` burns two seat NFTs and mints another, at the `totalSupply()` index.

```
function combineSeats(uint256 seatIdOne, uint256 seatIdTwo) public
returns(uint256) {
…
      _burn(seatIdOne);
      _burn(seatIdTwo);
      uint256 newSeatId = totalSupply();
      _safeMint(msg.sender, newSeatId);
      seatScore[newSeatId] = seatScore[seatIdOne] + seatScore[seatIdTwo];
      return seatScore[newSeatId];
}
```

However, the `totalSupply()` index is not guaranteed to not have been seen before. The totalSupply of an OpenZeppelin ERC721Enumerable is just the length of the enumerability array. When a token is being burned, it is removed from that array, its empty slot swapped with the last element, and the array gets truncated. Therefore, the above code will return as `newSeatId` an id that was previously used for different purposes. Although the `seatScore` is overwritten in the code, other data (namely, the `totalVolumeBySeat`) are not, and their old values will be confused with new.

The resolution of this issue (possibly by overriding function `_beforeTokenTransfer` to avoid reusing numbers) should be thoroughly tested, specifically by checking the indexes of old/new seatIds. It is unclear to us where the enumerability of NFTs functionality is used anyway. (This may mean that we are missing a potential threat in external use of ids that relates to the above behavior or its fix.)

| | | |
|---|---|---|
| M2 | Inconsistent, probably erroneous, fees in a Bid | **DISMISSED** |

The calculation of fees in Bid::fill assumes that the caller (i.e., the option seller that agrees to the Bid) has already factored the cost of fees into the `amountOfOPTION` argument. Specifically, the amount of options that the bid initiator/creator receives is exactly what the caller of `fill` has specified in `amountOfOPTION` but the Bid creator's USDC is supposed to cover the cost of both these options (per the option's `OPTIONPerUSDC` factor) and the fees.

```
function fill(uint256 amountOfOPTION, uint256 seatId) public nonReentrant {
…
  if(msg.sender == AORISEATSADD.ownerOf(seatId)) {
    …
  } else {
    //No taker fees are paid in option tokens, but rather USDC.
    OPTIONAfterFee = amountOfOPTION;
    //And the amount of the quote currency the msg.sender will receive
    USDCToReceive = mulDiv(OPTIONAfterFee, USDCDecimals, OPTIONPerUSDC);
    …
    USDC.transfer(Ownable(factory).owner(), ownerTxFee);
    USDC.transfer(AORISEATSADD.ownerOf(seatId), seatTxFee);
    USDC.transfer(msg.sender, USDCToReceive);
    //Tracking the liquidity mining rewards
    AORISEATSADD.addTakerPoints(feeMultiplier * (ownerTxFee / decimalDiff),
      msg.sender, factory);
    AORISEATSADD.addTakerPoints(feeMultiplier * (seatTxFee / decimalDiff),
      AORISEATSADD.ownerOf(seatId), factory);
    //Tracking the volume in the NFT
    AORISEATSADD.addTakerVolume(USDCToReceive, seatId, factory);
  }
…
}
```

This can be argued to be a design decision, but it has several surprising/inconsistent consequences:

- It puts a burden on external callers to do this calculation or risk reverting due to insufficient USDC in the contract.

- The taker of a Bid pays the fees, but the *maker* of a Bid gets the points, per the above `addTakerPoints` call! This is an asymmetry with Ask: whoever pays fees is likely expecting to get points.
- Another asymmetry with Asks is that, in an Ask, the above `addTakerVolume` calculation includes the USDC spent on fees. Here it does not.

| M3 | Counter-intuitive fee distribution in Asks and Bids | **RESOLVED (commit ccbf56c)** |
|---|---|---|

`Bid::fill` features the following logic (analogous logic can be found in `Ask::fill`):

```
…
    if(msg.sender == AORISEATSADD.ownerOf(seatId)) {
            …
    } else {
            //No taker fees are paid in option tokens, but rather USDC.
            OPTIONAfterFee = amountOfOPTION;
            //And the amount of the quote currency the msg.sender will receive
            USDCToReceive = mulDiv(OPTIONAfterFee, USDCDecimals, OPTIONPerUSDC);
//1eY = (1eX * 1eY) / 1eX

            //What the user will receive out of 100 percent in referral fees with
a floor of 40
            uint256 refRate = (AORISEATSADD.getSeatScore(seatId) * 5) + 35;
            //This means for Aori seat governance they should not allow more than
12 seats to be combined at once
            uint256 seatScoreFeeInBPS = mulDiv(fee, refRate, 100);
            uint256 ownerTxFee = mulDiv(USDCToReceive, seatScoreFeeInBPS, 10000);
            uint256 seatTxFee = mulDiv(USDCToReceive, fee - seatScoreFeeInBPS,
10000);
            //Transfers from the msg.sender
            OPTION.transferFrom(msg.sender, seller, OPTIONAfterFee);

            //Fee transfers are all in USDC, so for Bids they're routed here
            //These are to the Factory, the Aori seatholder, then the buyer
respectively.
            USDC.transfer(Ownable(factory).owner(), ownerTxFee);
            USDC.transfer(AORISEATSADD.ownerOf(seatId), seatTxFee);
```

```
            USDC.transfer(msg.sender, USDCToReceive);
            …
}
```

In principle, the higher the seat score, the larger the fee rewards that the `seatId` owner should receive. In the above case, however, a `seatId` with a higher score will receive fewer fees than a `seatId` with a lower seat score, simply because `fee - seatScoreFeeInBPS` will represent a larger value in the case of a lower seat score.

Additionally, the owner of the `Orderbook` contract will be the one receiving the seat fees, while the owner of the seat will receive whatever is left. This is asymmetrical with the logic that `AoriPut::mintPut` and `AoriCall::mintCall` implement:

```
…
            //If the owner of the seat is not the caller, calculate and transfer
the fees
            mintingFee = putUSDCFeeCalculator(quantityOfUSDC,
AORISEATSADD.getOptionMintingFee());
            uint256 refRate = (AORISEATSADD.getSeatScore(seatId) * 5) + 35;
            // Calculating the fees out of 100 to go to the seat owner
            feeToSeat = (refRate * mintingFee) / 100;
            optionsToMint = ((quantityOfUSDC - mintingFee) * 10**USDC.decimals())
/ strikeInUSDC; //(1e6*1e6) / 1e6
            optionsToMintScaled = optionsToMint * decimalDiff;

            //transfer the USDC and route fees
            USDC.transferFrom(msg.sender, address(this), optionsToMint);
            USDC.transferFrom(msg.sender, Ownable(factory).owner(), mintingFee -
feeToSeat);
            USDC.transferFrom(msg.sender, AORISEATSADD.ownerOf(seatId),
feeToSeat);
…
```

The above-mentioned points imply that perhaps the fees of the seat owner and the owner of the `Orderbook` contract are inverted for both Ask::fill and Bid::fill.

| M4 | The functionality of `AoriCall::sellerSettlementITM` (and `AoriPut::sellerSettlementITM`) breaks under intended parameters | **RESOLVED (commit ccbf56c)** |
|---|---|---|

Both implementations use the wrong inequality between `optionsToSettle` and `optionsSold` (>= should be used instead)

```
function sellerSettlementITM(uint256 optionsToSettle) public nonReentrant returns
(uint256) {
…
       uint256 optionsSold = optionSellers[msg.sender];
…
       require(optionsSold > 0 && optionsSold <= optionsToSettle);
       require(settlementPrice > strikeInUSDC && hasEnded == true);

       uint256 UNDERLYINGToReceive = ((strikeInUSDC * USDC.decimals()) /
settlementPrice) * optionsSold; // (1e6*1e6/1e6) * 1e18
       //store the settlement
       uint256 newOptionsSold = optionsSold - optionsToSettle;
       optionSellers[msg.sender] = newOptionsSold;

       //settle
       UNDERLYING.transfer(msg.sender, UNDERLYINGToReceive / 10**USDC.decimals());
…
}
```

Both `AoriCall::sellerSettlementITM` and `AoriPut::sellerSettlementITM` will revert if the seller chooses to settle fewer options than his `optionSellers` balance, breaking part of the intended functionality.

Additionally, `AoriCall::sellerSettlementITM` (the code snippet above) should be computing `UNDERLYINGToReceive` by multiplying with `optionsToSettle` and not `optionsSold`

## LOW SEVERITY:

| ID | Description | STATUS |
|---|---|---|
| L1 | View function can revert, possibly causing UI problems | **PARTLY RESOLVED** |

View functions Ask::getAmountFilled and Bid::getAmountFilled will revert if an attacker sends (even a tiny amount of) extra tokens to the contract (via a direct transfer). This is not a problem at the level of the contract, but could render an unsuspecting UI unusable until there is human intervention, thus causing an effective DoS for little cost.

```
function getCurrentBalance() public view returns (uint256) {
      return USDC.balanceOf(address(this));
}
function getAmountFilled() public view returns (uint256) {
      return (USDCSize - getCurrentBalance());
}
```

Similarly, the amounts that these functions return are not to be fully trusted by external agents, as they can be lower than actual.

| | | |
|---|---|---|
| L2 | `Ask::fundContract` (and `Bid::fundContract`) feature a questionable design | **(LARGELY) RESOLVED** |

While we did not find direct consequences in terms of security, the implementation of `Ask::fundContract` (and `Bid::fundContract`) raises questions on whether the intended functionality behind the funding of an Ask or a Bid has been fully thought of.

```
function fundContract() public nonReentrant {
      require(msg.sender == seller);
      require(OPTION.balanceOf(msg.sender) >= OPTIONSize);
      OPTION.transferFrom(msg.sender, address(this), OPTIONSize);
      startingBlock = block.number;
      endingBlock = block.number + duration;
      emit OfferFunded(seller, OPTIONSize, duration);
```

```
}
```

We took note of the following points:

- This function can be called multiple times
- Anyone may fund an Ask by directly sending `OPTION` (or `USDC` in the case of a Bid) tokens to the contract. This has the additional effect of making `OPTIONSize` (and `USDCSize`) simply serve as a minimum.

| | | |
|---|---|---|
| L3 | `AoriCall::getPrice` (and `AoriPut::getPrice`) do not perform staleness checks on the round data received by the Chainlink Aggregator | **RESOLVED (commits 5338e86, 8a74178)** |

Even though the `AggregatorV3::latestRoundData` function provides various return values that can be used to check the staleness of an answer (e.g., as the result of oracle downtime or the update round being incomplete at the time of querying), no such checks are performed.

```
function latestRoundData() external view returns (
      uint80 roundId,
      int256 answer,
      uint256 startedAt,
      uint256 updatedAt, //will be 0 if the round is incomplete
      uint80 answeredInRound
);
```

This can certainly undermine the experience of protocol users, as options will be settled based on stale prices.

Prolonged periods of down-time for most of the USDC denominated data-feeds are not likely, but in that scenario there could be direct consequences in terms of the protocol security.

| | | |
|---|---|---|
| L4 | Data feed answers that are either negative or zero are not handled consistently | **RESOLVED (commits d343c42,** |

| | | 4aa163c) |
|---|---|---|

In principle, the answer that is provided by a Chainlink Aggregator can be ≤ 0, but this is not consistently handled throughout AoriCall and AoriPut contracts.

Negative answers will cause the settlement price to be extremely large because it will have been cast to an `uint256`.

For `AoriPut`, price answers which are 0 will be silently accepted as settlement prices if `AoriPut::sellerSettlementITM` gets called first (it could be even called with `optionsToSettle` being 0)

```
function sellerSettlementITM(uint256 optionsToSettle) public nonReentrant returns
(uint256) {
      _setSettlementPrice();
      uint256 optionsSold = optionSellers[msg.sender];
...
      require(optionsSold > 0 && optionsSold <= optionsToSettle);
      require(strikeInUSDC > settlementPrice && hasEnded == true);

      uint256 USDCToReceive = ((optionsToSettle / decimalDiff) * settlementPrice)
/ 10**USDC.decimals(); //((1e18 / 1e12) * 1e6) / 1e6
...
      uint256 newOptionsSold = optionsSold - optionsToSettle;
      optionSellers[msg.sender] = newOptionsSold;
...
}
```

In this scenario, the buyer will never be able to settle in-the-money as all calls to `AoriPut::buyerSettlementITM` will revert

```
function buyerSettlementITM(uint256 optionsToSettle) public nonReentrant
returns (uint256) {
      _setSettlementPrice();
      require(block.number >= endingBlock && balanceOf[msg.sender] >= 0);
```

```
        require(strikeInUSDC > settlementPrice && settlementPrice != 0);
...
}
```

## CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

| ID | Description | STATUS |
|---|---|---|
| N1 | Some entities are considered trusted | **INFO** |
| The protocol has some centralization risks, with some `owner` entities considered trusted. For instance, the owner of an Orderbook can claim any tokens (including Options) from any Ask or Bid; the owner of an OrderbookFactory can change external contract addresses that implement significant functionality. | | |

## OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

| ID | Description | STATUS |
|---|---|---|
| A1 | Arithmetic may lose precision | **RESOLVED** |

Although the problem is likely very limited, given the expected magnitudes of the numbers, the following arithmetic in AoriCall::mintCall will maintain higher precision with the multiplication performed before the division.

```
AORISEATSADD.addPoints(
 feeMultiplier * ((mintingFee - feeToSeat) / decimalDiff), msg.sender);
AORISEATSADD.addPoints(
 feeMultiplier * (feeToSeat / decimalDiff), AORISEATSADD.ownerOf(seatId));
```

The same applies to the following arithmetic operation in `AoriPut::sellerSettlementITM`

```
...
      uint256 USDCToReceive = ((optionsToSettle / decimalDiff) *
settlementPrice) / 10**USDC.decimals();

...
      uint256 newOptionsSold = optionsSold - optionsToSettle;
      optionSellers[msg.sender] = newOptionsSold;

      //settle
      USDC.transfer(msg.sender, USDCToReceive);
...
```

In the last snippet, `USDCToReceive` can end up being zero when `optionsToSettle < decimalDiff`, in which case the `optionSellers` balance of the seller will be reduced but without the seller receiving any `USDC` in return.

| A2 | Unneeded generality? | **RESOLVED** |
|---|---|---|

In both AoriPut and AoriCall, it is not clear why the `sellerSettlementITM` function should allow settling fewer than all the seller's options. There is no financial sense in doing so: the loss of the option seller is known and is independent of the specifics of

each buyer. If the seller gets a refund for one buyer's options, they might as well get it for all their options, as they stand to gain nothing more by waiting.

| A3 | Unused storage variables | **RESOLVED** |
|---|---|---|

In AoriSeats, storage variables `seatPrice`, `startingIndex`, and `startingIndexBlock` are unused. The same is true of storage variable `ORDERBOOK`, which is also misleading, since there will not be a single orderbook.

| A4 | Wasteful storage dereferences | **RESOLVED** |
|---|---|---|

Some references to storage can be avoided, for gas savings. For instance, in AoriSeats::combineSeats:

```
function combineSeats(uint256 seatIdOne, uint256 seatIdTwo) public
returns(uint256) {
  …
  require(seatScore[seatIdOne] + seatScore[seatIdTwo] <= maxSeatScore);
  …
  seatScore[newSeatId] = seatScore[seatIdOne] + seatScore[seatIdTwo];
}
```

could be rewritten as:

```
function combineSeats(uint256 seatIdOne, uint256 seatIdTwo) public
returns(uint256) {
  …
  uint256 newSeatScore = seatScore[seatIdOne] + seatScore[seatIdTwo];
  require(newSeatScore <= maxSeatScore);
  …
  seatScore[newSeatId] = newSeatScore;
}
```

The latter avoids three SLOAD instructions.

| A5 | Confusing term: "seller" | **RESOLVED** |
|---|---|---|

In the Bid contract, calling the initiator of a bid the "`seller`" is confusing and inconsistent with other uses of the term throughout. Specifically, the initiator of a Bid is the eventual *buyer* of the option, not its seller.

| A6 | Several variables should be immutable, saving gas and preventing updates upon code changes | **PARTLY RESOLVED** |
|---|---|---|

Many storage variables never change after construction and should be declared `immutable`, so they can be inlined as constants. (Some storage variables can even be declared `constant`, for compile-time inlining.) These include at least:

- Optiontroller: `USDC`, `AORISEATSADD`
- AoriCall: `oracle`, `AORISEATSADD`
- AoriPut: `USDC`, `AORISEATSADD`
- AoriAuctionHouse: `weth`, `duration`
- Orderbook: `USDC`, `fee_`, `OPTION`
- Ask/Bid: `AORISEATSADD`, `USDC`, `OPTION`, `OPTIONDecimals`, `USDCDecimals`, `decimalDiff`.

| A7 | Some (repeated) external calls can be eliminated for gas savings | **RESOLVED** |
|---|---|---|

Some external calls can be optimized.

- Calls to `USDC.decimals()` (AoriCall, AoriPut) can be performed once and stored in an immutable variable.
- Calls to `Ownable(factory).owner()` (twice in Ask::withdrawTokens) can be performed once and stored in a local variable.

| A8 | Boolean handling is inelegant | **PARTLY RESOLVED** |
|---|---|---|

Several boolean operations are inelegant and can be simplified for a more professional code look.

```
// AoriSeats::addTakerPoints
require(OPTIONTROLLER.checkIsOrder(Orderbook_, msg.sender) == true);
->
require(OPTIONTROLLER.checkIsOrder(Orderbook_, msg.sender));

// Ask::cancel, Bid::cancel
// (This code is also unnecessary, covered in an earlier require)
isFunded() == true
->
isFunded()

// Ask::isFunded, similar in Ask::isFundedOverOne,
// Bid::isFunded, Bid::isFundedOverOne
if (OPTION.balanceOf(address(this)) > 0)
{ return true; } else { return false; }
->
return (OPTION.balanceOf(address(this)) > 0);

// Optiontroller::checkIsOrder
checkIsListedOrderbook(Orderbook_) == true
->
checkIsListedOrderbook(Orderbook_)
```

| A9 | Magic constants in the code | **PARTLY RESOLVED** |
|---|---|---|

Ideally, numeric constants should be visible prominently at the top of a contract, instead of being buried in the code, for easier maintainability and readability.

There are several instances in the code where we would recommend giving a name to the constant so that it is prominently visible.

```
// AoriAuctionHouse::_safeTransferETH
to.call{ value: value, gas: 30_000 }(new bytes(0));

// AoriCall::mintCall, AoriPut::mintPut
refRate = (AORISEATSADD.getSeatScore(seatId) * 5) + 35;

// AoriCall::callUNDERLYINGFeeCalculator
require(UNDERLYING.decimals() == 18);
uint256 txFee = (optionsToSettle * fee) / 10000;

// AoriPut::putUSDCFeeCalculator
uint256 txFee = (quantityOfUSDC * fee) / 10000;

// AoriCall::getPrice
return (uint256(price) / (10**8 - 10**USDC.decimals()));

// AoriPut::getPrice
return (uint256(price) / 1e2);

// AoriSeats::mintSeat
if (currentSeatId % 10 == 0) {

// Ask::fill, Bid::fill
uint256 refRate = (AORISEATSADD.getSeatScore(seatId) * 5) + 35;
```

| A10 | Unnecessary code | LARGELY RESOLVED |
|---|---|---|

Several pieces of code are logically unnecessary or even dead.

In AoriPut::sellerSettlementITM:

```
require(USDCToReceive <= USDC.balanceOf(address(this)),
        "Not enough USDC in contract");
```

No similar check occurs elsewhere in the code, and the check is unnecessary because the subsequent transfer would revert anyway.

In Ask:

```
function withdrawTokens(address token) public {
    require(msg.sender == Ownable(factory).owner());
    if (token == 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE) {

payable(Ownable(factory).owner()).transfer(address(this).balance);
    } else {
            uint256 balance = IERC20(token).balanceOf(address(this));
            safeTransfer(token, Ownable(factory).owner(), balance);
    }
}

function emergencyRetreival(address token) public {  // YS:! spell
    require(msg.sender == Ownable(factory).owner());
    IERC20(token).transfer(Ownable(factory).owner(),
    IERC20(token).balanceOf(address(this)));
}
```

The second function (also: misspelling in name) is unnecessary, since it is subsumed by the first, and even more completely (handling the case of tokens that don't implement a modern `transfer`).

In Ask::fundContract and Bid::fundContract, this assignment is dead code:

```
      startingBlock = block.number;
```

In Ask::fill (similarly in Bid::fill), if the code does not change, the introduction and use of an always-zero variable seems pointless:

```
        uint256 txFee = 0;
        USDCAfterFee = (amountOfUSDC - txFee);
```

| A11 | View functions return unnecessarily large arrays | **DISMISSED** |
|---|---|---|

All Orderbook::getActive* view functions return arrays that are larger than necessary, with zero items at the end. For example:

```
function getActiveBids() public view returns (Bid[] memory) {
        Bid[] memory activeBids = new Bid[](bids.length);
        uint256 count;
        for (uint256 i; i < bids.length; i++) {
                Bid bid = Bid(bids[i]);
                if (bid.isFunded() && !bid.hasEnded()) {
                        activeBids[count++] = bid;
                }
        }

        return activeBids;
}
```

External callers should be aware of this convention and not rely on the array length.

| A12 | ERC20 `transfer/transferFrom` definitions do not handle old tokens | **RESOLVED** |
|---|---|---|

In AoriCall the code uses calls to the `IERC20 transfer` and `transferFrom` functions, e.g., in:

```
function sellerSettlementOTM() public nonReentrant returns (uint256) {
```

```
…
      UNDERLYING.transfer(msg.sender, optionsSold);
…
}
```

As well as in:

```
function mintCall(uint256 quantityOfUNDERLYING, uint256 seatId) public
nonReentrant returns (uint256) {
            ...
            UNDERLYING.transferFrom(msg.sender, address(this),
quantityOfUNDERLYING);
            ...
}
```

The definition of the `transfer` and `transferFrom` functions used in the contract (from the OpenZeppelin libraries) expects a boolean return value:

```
function transfer(address to, uint256 amount) external returns (bool);
```

```
function transferFrom(address from, address to,uint256 amount) external
returns (bool);
```

However, old tokens (most notably USDT—the highest-capitalization ERC-20 token) predate the ERC-20 token specification and support a definition of `transfer` and `transferFrom` that does not return anything. Therefore, the current code will revert if used with USDT as the underlying token. However, because it is a stablecoin, we do not expect it to be used as the underlying token of call options.

| A13 | Compiler bugs | **INFO** |
|---|---|---|

The code has the compile pragmas 0.8.11^ or 0.8.13^. For deployment, we recommend no floating pragmas, i.e., a specific version, so as to be confident about the baseline guarantees offered by the compiler. Versions 0.8.11 and 0.8.13, in particular, have some

known bugs, which we do not believe to affect the correctness of the contracts.

## DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

## ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.