SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	
Prepared by:	Sherlock
Lead Security Expert:	xiaoming90
Dates Audited:	March 13 - March 20, 2023
Prepared on:	March 25, 2023

Introduction

Bond Protocol is the next evolution of bonds-as-a-service. Olympus-style bonds have revolutionized the way protocols approach acquiring assets.

Scope

The contracts in-scope for this audit are:

```
src
 bases
|    BondBaseOSDA.sol
|    BondBaseOFDA.sol
|    BondBaseFPA.sol
|    BondBaseOracle.sol
 interfaces
|    AggregatorV2V3Interface.sol
|    IBondOSDA.sol
|    IBondOFDA.sol
|    IBondFPA.sol
|    IBondOracle.sol
|    IBondBatchAuctionFactoryV1.sol
|    IBondBatchAuctionV1.sol
|    IGnosisEasyAuction.sol
 BondFixedExpiryOSDA.sol
 BondFixedTermOSDA.sol
 BondFixedExpiryOFDA.sol
 BondFixedTermOSDA.sol
 BondFixedExpiryFPA.sol
 BondFixedTermFPA.sol
 BondChainlinkOracle.sol
 BondBatchAuctionFactoryV1.sol
 BondBatchAuctionV1.sol
```

The in-scope contracts integrate with these previously audited contracts:

```
src
 bases
|    BondBaseTeller.sol
|    BondBaseCallback.sol
 interfaces
|    IBondTeller.sol
|    IBondFixedExpiryTeller.sol
|    IBondFixedTermTellers.sol
|    IBondAggregator.sol
|    IBondCallback.sol
|    IBondAuctioneer.sol
```

```
BondAggregator.sol
BondFixedTermTeller.sol
BondFixedExpiryTeller.sol
BondSampleCallback.sol
ERC20BondToken.sol
```

The following sections provide context and details about the different implementations.

Findings

Each issue has an assigned severity:

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

Issues found

Medium	High
8	1

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

xiaoming90	usmannk	spyrosonic10
Bauer	whitehat	martin
Avci	Breeje	Diana

Issue H-1: "Equilibrium price" is not used to compute the capacity (OSDA Only)

Source: https://github.com/sherlock-audit/2023-02-bond-judging/issues/18

Found by

xiaoming90, Bauer

Summary

"Equilibrium price" is not used to compute the capacity leading to a smaller-than-expected max payout.

Vulnerability Detail

In OFDA, it was observed that if the capacity is denominated in the quote token, the capacity will be calculated with the discounted price.

```
File: BondBaseOFDA.sol
118:     function _createMarket(MarketParams memory params_) internal returns
 ↪  (uint256) {
..SNIP..
178:         // Calculate the maximum payout amount for this market
179:         uint256 capacity = params_.capacityInQuote
180:             ? params_.capacity.mulDiv(
181:                 scale,
182:                 price.mulDivUp(
183:                     uint256(ONE_HUNDRED_PERCENT - params_.fixedDiscount),
184:                     uint256(ONE_HUNDRED_PERCENT)
185:                 )
186:             )
187:             : params_.capacity;
188:         market.maxPayout =
 ↪  capacity.mulDiv(uint256(params_.depositInterval), uint256(length));
```

However, in OSDA, if the capacity is denominated in the quote token, the capacity will be calculated with the oracle price instead of the discounted price.

```
File: BondBaseOSDA.sol
122:     function _createMarket(MarketParams memory params_) internal returns
 ↪  (uint256) {
..SNIP..
182:         // Calculate the maximum payout amount for this market, determined
 ↪  by deposit interval
183:         uint256 capacity = params_.capacityInQuote
```

```
            ? params_.capacity.mulDiv(scale, price)
            : params_.capacity;
        market.maxPayout =
↪  capacity.mulDiv(uint256(params_.depositInterval), uint256(length));
```

In OSDA, it was also observed that the base discount is applied to the oracle price while calculating the price decimals because this will be the initial equilibrium price of the market. However, this "initial equilibrium price" is not used earlier when computing the capacity.

```
File: BondBaseOSDA.sol
    function _validateOracle(
        uint256 id_,
        IBondOracle oracle_,
        ERC20 quoteToken_,
        ERC20 payoutToken_,
        uint48 baseDiscount_
    )
..SNIP..
        // Get the price decimals for the current oracle price
        // Oracle price is in quote tokens per payout token
        // E.g. if quote token is $10 and payout token is $2000,
        // then the oracle price is 200 quote tokens per payout token.
        // If the oracle has 18 decimals, then it would return 200 * 10^18.
        // In this case, the price decimals would be 2 since 200 = 2 * 10^2.
        // We apply the base discount to the oracle price before calculating
        // since this will be the initial equilibrium price of the market.
        int8 priceDecimals = _getPriceDecimals(
            currentPrice.mulDivUp(
                uint256(ONE_HUNDRED_PERCENT - baseDiscount_),
                uint256(ONE_HUNDRED_PERCENT)
            ),
            oracleDecimals
        );
```

Impact

As the discount is not applied to the price when computing the capacity, the price will be higher which leads to a smaller capacity. A smaller capacity will in turn result in a smaller max payout. A smaller-than-expected max payout reduces the maximum number of payout tokens a user can purchase at any single point in time, which might reduce the efficiency of a Bond market.

Users who want to purchase a large number of bond tokens have to break their trade into smaller chunks to overcome the smaller-than-expected max payout, leading to unnecessary delay and additional gas fees.

Code Snippet

https://github.com/sherlock-audit/2023-02-bond/blob/main/bonds/src/bases/BondBaseOSDA.sol#L122

Tool used

Manual Review

Recommendation

Applied the discount to obtain the "equilibrium price" before computing the capacity.

```
// Calculate the maximum payout amount for this market, determined by deposit
↪  interval
uint256 capacity = params_.capacityInQuote
-    ? params_.capacity.mulDiv(scale, price)
+    ? params_.capacity.mulDiv(scale, price.mulDivUp(
+           uint256(ONE_HUNDRED_PERCENT - params_.baseDiscount),
+           uint256(ONE_HUNDRED_PERCENT)
+       )
+    )
    : params_.capacity;
market.maxPayout = capacity.mulDiv(uint256(params_.depositInterval),
↪  uint256(length));
```

Issue M-1: The createMarket transaction lack of expiration timestamp check

Source: https://github.com/sherlock-audit/2023-02-bond-judging/issues/60

Found by

whitehat

Summary

The createMarket transaction lack of expiration timestamp check

Vulnerability Detail

Let us look into the heavily forked Uniswap V2 contract addLiquidity function implementation

https://github.com/Uniswap/v2-periphery/blob/0335e8f7e1bd1e8d8329fd300aea2ef2f36dd19f/contracts/UniswapV2Router02.sol#L61

```
// **** ADD LIQUIDITY ****
function _addLiquidity(
    address tokenA,
    address tokenB,
    uint amountADesired,
    uint amountBDesired,
    uint amountAMin,
    uint amountBMin
) internal virtual returns (uint amountA, uint amountB) {
    // create the pair if it doesn't exist yet
    if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
        IUniswapV2Factory(factory).createPair(tokenA, tokenB);
    }
    (uint reserveA, uint reserveB) = UniswapV2Library.getReserves(factory,
↪   tokenA, tokenB);
    if (reserveA == 0 && reserveB == 0) {
        (amountA, amountB) = (amountADesired, amountBDesired);
    } else {
        uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA,
↪   reserveB);
        if (amountBOptimal <= amountBDesired) {
            require(amountBOptimal >= amountBMin, 'UniswapV2Router:
↪   INSUFFICIENT_B_AMOUNT');
            (amountA, amountB) = (amountADesired, amountBOptimal);
        } else {
            uint amountAOptimal = UniswapV2Library.quote(amountBDesired,
↪   reserveB, reserveA);
            assert(amountAOptimal <= amountADesired);
            require(amountAOptimal >= amountAMin, 'UniswapV2Router:
↪   INSUFFICIENT_A_AMOUNT');
            (amountA, amountB) = (amountAOptimal, amountBDesired);
        }
    }
}

function addLiquidity(
    address tokenA,
    address tokenB,
    uint amountADesired,
    uint amountBDesired,
    uint amountAMin,
    uint amountBMin,
    address to,
    uint deadline
) external virtual override ensure(deadline) returns (uint amountA, uint
↪   amountB, uint liquidity) {
    (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,
↪   amountBDesired, amountAMin, amountBMin);
    address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);
    TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
    TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
    liquidity = IUniswapV2Pair(pair).mint(to);
}
```

the implementation has two point that worth noting,

the first point is the deadline check

```
modifier ensure(uint deadline) {
    require(deadline >= block.timestamp, 'UniswapV2Router: EXPIRED');
    _;
}
```

The transaction can be pending in mempool for a long time and can be executed in a long time after the user submit the transaction.

Problem is createMarket, which calculates the length and maxPayout by block.timestamp inside it.

```
// Calculate market length and check time bounds
uint48 length = uint48(params_.conclusion - block.timestamp); \
if (
    length < minMarketDuration ||
    params_.depositInterval < minDepositInterval ||
    params_.depositInterval > length
) revert Auctioneer_InvalidParams();

// Calculate the maximum payout amount for this market, determined by deposit
↪  interval
uint256 capacity = params_.capacityInQuote
    ? params_.capacity.mulDiv(scale, price)
    : params_.capacity;
market.maxPayout = capacity.mulDiv(uint256(params_.depositInterval),
↪  uint256(length));
```

After the market is created at wrong time, user can call purchase. At purchaseBond(),

```
// Payout for the deposit = amount / price
//
// where:
// payout = payout tokens out
// amount = quote tokens in
// price = quote tokens : payout token (i.e. 200 QUOTE : BASE), adjusted for
↪  scaling
payout = amount_.mulDiv(term.scale, price);

// Payout must be greater than user inputted minimum
if (payout < minAmountOut_) revert Auctioneer_AmountLessThanMinimum();

// Markets have a max payout amount, capping size because deposits
// do not experience slippage. max payout is recalculated upon tuning
if (payout > market.maxPayout) revert Auctioneer_MaxPayoutExceeded();
```

payout value is calculated by term.scale which the market owner has set assuming the market would be created at desired timestamp. Even, maxPayout is far bigger than expected, as it is calculated by very small length.

Impact

Even though the market owner close the market at any time, malicious user can attack the market before close and steal unexpectedly large amount of payout Tokens.

Code Snippet

Tool used

Manual Review

Recommendation

Use deadline, like uniswap

Discussion

0ighty

Agree with this finding. We have noticed some issues with shorter than expected durations for existing markets.

Our proposed fix is to have users specify a `start` timestamp and a `duration`, which will be used to calculate/set the conclusion. If `block.timestamp` is greater than the `start`, then the txn will revert. Therefore, users must create the market before the target start time. We may allow this to be bypassed by providing a start time of zero, which would then start the market at the `block.timestamp` for the provided `duration`.

hrishibhat

Given the pre-condition that the transaction needs to be in the mempool for a long time for it to have a significant impact, considering this issue as valid medium

Issue M-2: Multiplication after Division can cause larger Precision loss

Source: https://github.com/sherlock-audit/2023-02-bond-judging/issues/49

Found by

Breeje

Summary

There are multiple instances in the code where multiplication is done on the result of division.

Vulnerability Detail

In several critical calculations like calculating the payouts or current market price, multiplication is done on the result of division which can lead to larger precision loss.

Instances:

1. `payoutFor` method in `BondBaseOSDA.sol`:

```
File: BondBaseOSDA.sol

548:    uint256 fee = amount_.mulDiv(_teller.getFee(referrer_), 1e5);
549:    uint256 payout = (amount_ - fee).mulDiv(terms[id_].scale, marketPrice(id_));
```

Link to Code

2. `_currentMarketPrice` method in `BondBaseOSDA.sol`:

```
File: BondBaseOSDA.sol

Instance 1:

431:    uint256 price = term.oracle.currentPrice(id_).mulDivUp(
440:    price = price * term.oracleConversion;

Instance 2:

451:    uint256 expectedCapacity = initialCapacity.mulDiv(timeRemaining, uint256(term.length));
472:    (term.decaySpeed * (expectedCapacity - market.capacity)) /
```

```
    uint256 factor = (term.decaySpeed * (market.capacity - expectedCapacity)) /
    return price.mulDivUp(adjustment, ONE_HUNDRED_PERCENT);
```

Link to Code

3. `maxAmountAccepted` method in `BondBaseOFDA`:

```
File: BondBaseOFDA.sol

    uint256 quoteCapacity = market.capacityInQuote
        ? market.capacity
        : market.capacity.mulDiv(price, term.scale);
    uint256 maxQuote = market.maxPayout.mulDiv(price, term.scale);
    uint256 amountAccepted = quoteCapacity < maxQuote ? quoteCapacity : maxQuote;

    uint256 estimatedFee = amountAccepted.mulDiv(
```

Link to Code

Impact

Larger Precision Loss.

Code Snippet

Given above.

Tool used

Manual Review

Recommendation

Multiply all the numerators first and then divide it with the product of all the denominator to get the least possible precision loss.

Discussion

Oighty

In general, I think this is mitigated by the scaling we use, but I'll try to address these individually.

1. The `fee` is calculated by a mulDiv with the feePercent multiplied first and then divided by 100%. We have to finish this calculation here because it is used in a subtraction on the next line.

2.1. This calculation could increase precision slightly by multiplying the `oracleConversion` prior to implementing the discount. Will fix this.

2.2. `expectedCapacity` has to be pre-computed because it's used in a subtraction operation. Additionally, there is an addition or subtraction in the `adjustment` calculation so the values must be pre-computed before the final operation. I don't think the precision of these can increase. Even if it did, the code would be much less legible.

3. I don't see how the precision can be improved because the `maxQuote` and `quoteCapacity` amounts have to be pre-computed to allow for the comparison to get `amountAccepted`. Perhaps if you did the comparison and then recalculated the right one, but the precision loss is negligible and the amount we're calculating here is conservative anyways.

Issue M-3: Removing support for a currency pair from the oracle leaves markets in an invalid state

Source: https://github.com/sherlock-audit/2023-02-bond-judging/issues/44

Found by

usmannk

Summary

When a new market is registered in the Bond Oracle, it is checked that that market's currencies are supported by the oracle. If not, then the transaction reverts.

However, when a currency pair is later set to be unsupported due to an issue, markets with that pair are not unregistered. This breaks the invariant that all registered markets use supported currencies.

In fact, there is actually no way for an owner to unregister a market at all.

Vulnerability Detail

Market operations on registered markets with unsupported tokens will attempts calls to `address(0)`, causing unexpected results for oracle operations

https://github.com/sherlock-audit/2023-02-bond/blob/main/bonds/src/BondChainlinkOracle.sol#L129-L148

Impact

Unexpected state for markets, potentially causing unexplained DoS.

Code Snippet

Tool used

Manual Review

Recommendation

Provide a mechanism for unregistering a market from the oracle. In addition, add auto-unregistering for markets using newly unsupported currency pairs.

Discussion

Oighty

Owners have the ability to close markets they have active on any Auctioneer. If an oracle removes support for a token pair, then the `currentPrice()` function will fail. Specifically, it will execute the path `_getOneFeedPrice` and then `_validateAndGetPrice`, which will revert when validating the empty data returned from the call to the zero address.

However, this is mostly by coincidence. I think the best approach is having `currentPrice()` revert if the priceFeedParams for the pair are bytes(0) since this will be the case immediately after the pair is no longer supported and provides clearer behavior.

Issue M-4: Possibility of underflow in calling `initiateBatchAuction`

Source: https://github.com/sherlock-audit/2023-02-bond-judging/issues/39

Found by

spyrosonic10, Bauer

Summary

BondBatchAuctionV1.sol has a function `amountWithFee()` which calculate amount with fee from given input and this function has a logic to check fee like this `_teller.protocolFee() - _teller.createFeeDiscount()`. There is no check to make sure `protocolFee` is higher than `createFeeDiscount` and hence has potential to fail if it is lower.

Vulnerability Detail

There are multiple instance where code does check `if (protocolFee > createFeeDiscount)` which indicate that it is possible for `protocolFee` to be lower than createFeeDiscount and in such cases failure will occur. Fee and discount are set in BondBaseTeller(which is not in scope) and it does suggest that there is no safe check in place during setting `setProtocolFee` which also validate possible existence of vulnerability.

Impact

If `protocolFee` is lower than `createFeeDiscount` then `amountWithFee` will fail and which cause `initiateBatchAuction` to fail.

Code Snippet

https://github.com/sherlock-audit/2023-02-bond/blob/main/bonds/src/BondBatchAuctionV1.sol#L306-L321

Tool used

Manual Review

Recommendation

Consider adding `if (protocolFee > createFeeDiscount)` before performing subtraction.

Discussion

Oighty

It's not supposed to be able to be greater. We do check that `createFeeDiscount` is not set to be greater than `protocolFee` in the `setCreateFeeDiscount` function on the BaseTeller, but I see that we didn't include a check on the `setProtocolFee` function to ensure it is greater than `createFeeDiscount`. Therefore, you could technically set it higher by setting in and then changing the `protocolFee`. Based on that, I agree it makes sense to add this check, but since it is a protocol parameter, the risk is low.

Issue M-5: Use call() instead of transfer() on an address payable

Source: https://github.com/sherlock-audit/2023-02-bond-judging/issues/37

Found by

Diana, martin, Bauer

Summary

The use of the deprecated transfer() function for an payable address will inevitably make the transaction

Vulnerability Detail

The transfer() and send() functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase of the SLOAD instruction.

```
function emergencyWithdraw(ERC20 token_) external override onlyOwner {
    // Confirm that the token is a contract
    if (address(token_).code.length == 0 && address(token_) != address(0))
        revert BatchAuction_InvalidParams();

    // If token address is zero, withdraw ETH
    if (address(token_) == address(0)) {
        payable(owner()).transfer(address(this).balance);
    } else {
        token_.safeTransfer(owner(), token_.balanceOf(address(this)));
    }
}
```

Impact

The use of the deprecated transfer() function for an address will inevitably make the transaction fail when:

The claimer smart contract does not implement a payable function. The claimer smart contract does implement a payable fallback which uses more than 2300 gas unit. The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300. Additionally, using higher than 2300 gas might be mandatory for some multisig wallets.

Code Snippet

https://github.com/sherlock-audit/2023-02-bond/blob/main/bonds/src/BondBatchAuctionV1.sol#L287

Tool used

Manual Review

Recommendation

Use call() instead of transfer(), but be sure to respect the CEI pattern and/or add re-entrancy guards, as several hacks already happened in the past due to this recommendation not being fully understood.

More info on; https://swcregistry.io/docs/SWC-134

Discussion

Oighty

Agree with making this change. I don't think CEI/re-entrancy guard is necessary since this function can only be called by the contract owner and sends the whole balance on the first call. No other state is changed.

Issue M-6: Inconsistencies within the `payoutFor` function

Source: https://github.com/sherlock-audit/2023-02-bond-judging/issues/26

Found by

xiaoming90

Summary

Inconsistencies within the `payoutFor` function were observed. The function does not revert when there is insufficient capacity for the number of payout tokens they are expected to receive when depositing a certain number of quote tokens. This gives the caller an incorrect state of the current market.

Vulnerability Detail

> This issue affects FPA, OFDA, and OSDA

Within the `purchaseBond` function, the function will revert, and the users will not be able to purchase the bond when any of the conditions is met:

- The payout exceeds the max payout
- Amount/payout is greater than the capacity remaining

```
File: BondBaseFPA.sol
    function purchaseBond(
        uint256 id_,
        uint256 amount_,
        uint256 minAmountOut_
    ) external override returns (uint256 payout) {
..SNIP
        // Markets have a max payout amount per transaction
        if (payout > market.maxPayout) revert Auctioneer_MaxPayoutExceeded();
..SNIP..
        // If amount/payout is greater than capacity remaining, revert
        if (market.capacityInQuote ? amount_ > market.capacity : payout > market.capacity)
            revert Auctioneer_NotEnoughCapacity();
        // Capacity is decreased by the deposited or paid amount
        market.capacity -= market.capacityInQuote ? amount_ : payout;
..SNIP..
    }
```

It was observed that there are some inconsistencies within the `payoutFor` function. The function will revert when the payout exceeds the max payout. However, it does not revert if the amount/payout is greater than the capacity remaining.

```
File: BondBaseFPA.sol
    function payoutFor(
        uint256 amount_,
        uint256 id_,
        address referrer_
    ) public view override returns (uint256) {
        // Calculate the payout for the given amount of tokens
        uint256 fee = amount_.mulDiv(_teller.getFee(referrer_), ONE_HUNDRED_PERCENT);
        uint256 payout = (amount_ - fee).mulDiv(markets[id_].scale, marketPrice(id_));

        // Check that the payout is less than or equal to the maximum payout,
        // Revert if not, otherwise return the payout
        if (payout > markets[id_].maxPayout) {
            revert Auctioneer_MaxPayoutExceeded();
        } else {
            return payout;
        }
    }
```

Assume the following:

- maxPayout = 100 Bond Tokens (DAI)
- market.capacity = 50 Bond Tokens (DAI) left
- payout = 70 Bond Tokens (DAI) should be issued to the users

Assume that Alice calls the `payoutFor` function for a certain amount of quote tokens. The function will return `70 Bond Tokens (DAI)` and will not revert because `payout` is lower than the `maxPayout`.

This makes Alice believe she can obtain `70 Bond Tokens (DAI)` if she calls the `purchaseBond` function. However, when she calls the `purchaseBond` function, the function reverts because of insufficient capacity.

Impact

Inconsistencies within the protocol can cause various issues, especially for the external or third-party protocol that relies on protocol's interfaces to determine the correct state of the protocol they are integrating with. Also, callers relying on the `payoutFor` result might wrongly assume that their `purchaseBond` transaction will go through when there is insufficient capacity, incurring unnecessary delay or missed opportunity during trading.

Code Snippet

https://github.com/sherlock-audit/2023-02-bond/blob/main/bonds/src/bases/BondBaseFPA.sol#L350

https://github.com/sherlock-audit/2023-02-bond/blob/main/bonds/src/bases/BondBaseOFDA.sol#L470

https://github.com/sherlock-audit/2023-02-bond/blob/main/bonds/src/bases/BondBaseOSDA.sol#L542

Tool used

Manual Review

Recommendation

Update the `payoutFor` function to revert when there is insufficient capacity.

```
function payoutFor(
    uint256 amount_,
    uint256 id_,
    address referrer_
) public view override returns (uint256) {
    // Calculate the payout for the given amount of tokens
    uint256 fee = amount_.mulDiv(_teller.getFee(referrer_), ONE_HUNDRED_PERCENT);
    uint256 payout = (amount_ - fee).mulDiv(markets[id_].scale,
↪   marketPrice(id_));

    // Check that the payout is less than or equal to the maximum payout,
    // Revert if not, otherwise return the payout
    if (payout > markets[id_].maxPayout) {
        revert Auctioneer_MaxPayoutExceeded();
+   } else if (market[id_].capacityInQuote ? amount_ > market[id_].capacity :
↪   payout > market[id_].capacity) {
+       revert Auctioneer_NotEnoughCapacity();
    } else {
        return payout;
    }
}
```

Discussion

Oighty

Agree with this finding. `payoutFor` should revert if the amount will exceed capacity and be consistent across the contracts. We will fix.

Issue M-7: Inconsistent max payout returned from `getMarketInfoFo` across different auctioneers

Source: https://github.com/sherlock-audit/2023-02-bond-judging/issues/25

Found by

xiaoming90

Summary

The max payout returned by `getMarketInfoForPurchase` function has a different meaning depending on the type of Auctioneer. Given these differences, any parties (internal or external) that rely on the `getMarketInfoForPurchase` function will have an issue because of the inconsistent result returned by the interface.

Vulnerability Detail

The `BondBaseFPA.getMarketInfoForPurchase` and `BondBaseOFDA.getMarketInfoForPurchase` functions retrieve the market's max payout directly from `market.maxPayout`.

```
File: BondBaseFPA.sol
    function getMarketInfoForPurchase(
        uint256 id_
    )
        external
        view
        returns (
            address owner,
            address callbackAddr,
            ERC20 payoutToken,
            ERC20 quoteToken,
            uint48 vesting,
            uint256 maxPayout
        )
    {
        BondMarket memory market = markets[id_];
        return (
            market.owner,
            market.callbackAddr,
            market.payoutToken,
            market.quoteToken,
            terms[id_].vesting,
            market.maxPayout
        );
    }
```

```
File: BondBaseOFDA.sol
    function getMarketInfoForPurchase(uint256 id_)
        external
        view
        returns (
            address owner,
            address callbackAddr,
            ERC20 payoutToken,
            ERC20 quoteToken,
            uint48 vesting,
            uint256 maxPayout
        )
    {
        BondMarket memory market = markets[id_];
        return (
            market.owner,
            market.callbackAddr,
            market.payoutToken,
            market.quoteToken,
            terms[id_].vesting,
            market.maxPayout
        );
    }
```

However, for the `BondBaseOSDA.getMarketInfoForPurchase` function, it was done differently. It calls the `BondBaseOSDA.maxPayout` function instead of accessing the `market.maxPayout` to obtain the max payout of a market.

```
File: BondBaseOSDA.sol
    function getMarketInfoForPurchase(uint256 id_)
        external
        view
        override
        returns (
            address owner,
            address callbackAddr,
            ERC20 payoutToken,
            ERC20 quoteToken,
            uint48 vesting,
            uint256 maxPayout_
        )
    {
        BondMarket memory market = markets[id_];
        return (
520:                 market.owner,
521:                 market.callbackAddr,
522:                 market.payoutToken,
523:                 market.quoteToken,
524:                 terms[id_].vesting,
525:                 maxPayout(id_)
526:             );
527:     }
```

The result returned from the `BondBaseOSDA.maxPayout` function is different from the `market.maxPayout` because if the capacity is lower than the max payout, the max payout will be capped at the remaining capacity. Refer to Line 573 below.

```
File: BondBaseOSDA.sol
561:     function maxPayout(uint256 id_) public view override returns (uint256) {
562:         // Get current price
563:         uint256 price = marketPrice(id_);
564:
565:         BondMarket memory market = markets[id_];
566:         BondTerms memory term = terms[id_];
567:
568:         // Convert capacity to payout token units for comparison with max
 ↪  payout
569:         uint256 capacity = market.capacityInQuote
570:             ? market.capacity.mulDiv(term.scale, price)
571:             : market.capacity;
572:
573:         // Cap max payout at the remaining capacity
574:         return market.maxPayout > capacity ? capacity : market.maxPayout;
575:     }
```

Assume the max payout of an FPA, OFDA, and OSDA market is 100, and their current capacity is 50.

Calling `BondBaseFPA.getMarketInfoForPurchase` and `BondBaseOFDA.getMarketInfoForPurchase` functions will return 100 as the max payout, while calling `BondBaseOSDA.getMarketInfoForPurchase` function will return 50 as the max payout.

Impact

The max payout returned by `getMarketInfoForPurchase` function has a different meaning depending on the type of Auctioneer. For FPA and OFDA, it means the max payout that is configured during market initialization, and it stays constant. For OSDA, it means the maximum amount of payout tokens that can be issued to a user at this point in time, and it is not constant and takes into consideration of the current capacity of the market.

Given these differences, any parties (internal or external) that rely on the `getMarketInfoForPurchase` function will have an issue because of the inconsistent result returned by the interface.

Code Snippet

https://github.com/sherlock-audit/2023-02-bond/blob/main/bonds/src/bases/BondBaseFPA.sol#L314

https://github.com/sherlock-audit/2023-02-bond/blob/main/bonds/src/bases/BondBaseOFDA.sol#L422

https://github.com/sherlock-audit/2023-02-bond/blob/main/bonds/src/bases/BondBaseOSDA.sol#L505

Tool used

Manual Review

Recommendation

Standardize the max payout returned from the `getMarketInfoForPurchase` function across all the auctioneers (FPA, OFDA, OSDA, SDA)

Discussion

Oighty

Agree that this should be updated to be consistent across auctioneers. We will use the function value since it incorporates the capacity as a limit. The `maxPayout` value from this function is currently only used in a view function in the Aggregator.

Issue M-8: _validateAndGetPrice() doesn't check If Arbitrum sequencer is down in Chainlink feeds

Source: https://github.com/sherlock-audit/2023-02-bond-judging/issues/1

Found by

Avci

Summary

When utilizing Chainlink in L2 chains like Arbitrum, it's important to ensure that the prices provided are not falsely perceived as fresh, even when the sequencer is down. This vulnerability could potentially be exploited by malicious actors to gain an unfair advantage.

Vulnerability Detail

There is no check:

Impact

could potentially be exploited by malicious actors to gain an unfair advantage.

Code Snippet

https://github.com/sherlock-audit/2023-02-bond-0xdanial/blob/0d6f979c9f361bc1101f429b3bb09264577b9a71/bonds/src/BondChainlinkOracle.sol#L129

Tool used

Manual Review

Recommendation

code example of Chainlink:
https://docs.chain.link/data-feeds/l2-sequencer-feeds#example-code

Discussion

Oighty

Agree this should be fixed for using the Chainlink Oracle Contract on L2s. I think the best way to handle is to have a mainnet version of the contract (as is) and L2 version of the contract which implements the sequencer feed check.