SECURITY REVIEW OF ARROW MARKETS V2

Summary

Auditors: 0xWeiss (Marc Weiss)

Client: Arrow Markets V2

Report Delivered: 29 January, 2024

About 0xWeiss

0xWeiss is an independent security researcher. In-house auditor/security engineer in Ambit Finance and Tapioca DAO. Security Researcher at Paladin Blockchain Security and ASR at Spearbit DAO. Reach out on Twitter @0xWeisss .

Protocol Summary

Arrow Markets is a next generation options trading platform powered by a novel request-for-execution (RFE) network. Ownership, transfer, and settlement are handled on-chain while competitive prices are provided through our network of participating market makers. Arrow Markets' UX is world class, positioning the platform to onboard the next wave of web3 options traders.

Protocol Name	Arrow Markets V2
Language	Solidity
Codebase	https://github.com/ArrowDFMs/arrow-rfq-product
Commit	dee585601f2009ee2104027fe8fc6a53be55cff6
Previous Audits	None
Test Coverage	Decent
Fuzz Testing	Yes (Foundry Fuzz tests)
Key Management	Good ✅
Centralization	Centralized

Audit Summary

Arrow Markets V2 engaged **0xWeiss** through Hyacinth to review the security of its codebase.

A 3 week time-boxed security assesment was performed.

At the end, there were 33 issues identified.

All findings have been recorded in the following report. Notice that the examined smart contracts are not resistant to internal exploit.

For a detailed understanding of risk severity, source code vulnerability, and potential attack vectors, refer to the complete audit report below.

Vulnerability Summary

Severity	Total	Pending	Acknowledged	Par. resolved	Resolved
HIGH	**8**	**0**	**1**	**0**	**7**
MEDIUM	**10**	**0**	**2**	**2**	**6**
LOW	**13**	**0**	**1**	**0**	**12**
INF	**2**	**0**	**0**	**0**	**2**

Severity Classification

Severity	Classification
HIGH	Exploitable, causing loss/manipulation of assets or data.
MEDIUM	Risk of future exploits that may or may not impact the smart contract execution.
LOW	Minor code errors that may or may not impact the smart contract execution.
INF	No impact issues. Code improvement

Methodology

The auditing process pays special attention to the following considerations:

● Testing the smart contracts against both common and uncommon attack vectors.

● Assessing the codebase to ensure compliance with current best practices and industry standards.

● Ensuring contract logic meets the specifications and intentions of the client.

● Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.

● Thorough line-by-line manual review of the entire codebase by industry experts.

AUDIT SCOPE

CONTRACTS

ID	File Path
APM	contracts/ArrowPositionManager.sol
AE	contracts/ ArrowEvents.sol
UFM	contracts/ UserFundsManager.sol
APD	contracts/ ArrowPositionDelegate.sol
AR	contracts/ ArrowRouter.sol
AERC	contracts/ tokens/AERC1155.sol
AU	contracts/ libraries/ArrowUtils.sol
AP	contracts/ proxy/ArrowProxy.sol
AO	contracts/ArrowOptions.sol
ARE	contracts/ ArrowRegistry.sol

ABSTRACT

AAO	/ abstracts/ArrowOptions/AbstractArrowOptions.sol
AAOS	/ abstracts/ArrowOptions/AbstractArrowOptionsStorage.sol
AAR	/abstracts/ArrowRegistry/AbstractArrowRegistry.sol
AAOS	/abstracts/ArrowOptions/AbstractArrowOptionsStorage.sol
AAPM	/ArrowPositionManager/AbstractArrowPositionManager
AUFM	/UserFundsManager/AbstractUserFundsManagerStorage
AAPD	/ArrowPositionManager/AbstractArrowPositionDelegate
AAPS	/APM/AbstractArrowPositionManagerStorage.sol
AUFM	/UserFundsManager/AbstractUserFundsManager.sol
ERR	contracts/abstracts/Errors.sol
PAR	contracts/abstracts/Parameters.sol

Findings and Resolutions

ID	Category	Severity	Status
AU-H1	Signature	**HIGH**	Resolved
AU-H2	Signature	**HIGH**	Resolved
APM-H1	Protocol Loss	**HIGH**	Resolved
APM-H2	User Loss	**HIGH**	Acknowledged
APD-H1	User Loss	**HIGH**	Resolved
APD-H2	User Loss	**HIGH**	Resolved
APD-H3	User Loss	**HIGH**	Resolved
AR-H1	User Loss	**HIGH**	Resolved
OPD-M1	Protocol Loss	**MEDIUM**	Resolved
OPD-M1	DoS	**MEDIUM**	Resolved
UFM-M1	User Loss	**MEDIUM**	Resolved
UFM-M2	User Loss	**MEDIUM**	Acknowledged
GLOBAL-M1	User Loss	**MEDIUM**	Resolved
GLOBAL-M2	User Loss	**MEDIUM**	Resolved
GLOBAL-M3	Architectural Error	**MEDIUM**	Resolved
GLOBAL-M4	Architectural Error	**MEDIUM**	Partially Resolved
APM-M1	Architectural Error	**MEDIUM**	Partially Resolved
AR-M1	Grieving	**MEDIUM**	Acknowledged
AU-L1	Protocol Loss	**LOW**	Acknowledged
UFM-L1	Input Validation	**LOW**	Resolved
APM-L1	Composability	**LOW**	Resolved
AE-L1	Access Control	**LOW**	Resolved
GLOBAL-L1	Un-used Imports	**LOW**	Resolved
GLOBAL-L2	Architectural Error	**LOW**	Resolved
AO-L1	Input Validation	**LOW**	Resolved

AERC1155-L1	Architectural Error	**LOW**	Resolved
AR-L1	Composability	**LOW**	Resolved
AR-L2	Ownership	**LOW**	Resolved
APD-L1	Input Validation	**LOW**	Resolved
APD-L2	Input Validation	**LOW**	Resolved
APD-L2	Input Validation	**LOW**	Resolved
OU-INF1	Natspec	**INF**	Resolved
GLOBAL-INF1	Gas	**INF**	Resolved

[AU-H1] Expired order signatures can be executed by sending a faulty deadline.

Severity	Category	Status
HIGH	Signature	Resolved

Description

Currently, the signatures are implemented in `ArrowUtils` . The signature that includes the order details fails to specify the deadline param `orderDeadline`, which is not included in the OrderData signature.

Users have to specify the deadline when opening a position from the router `uint256 orderDeadline; // order deadline - Unix timestamp (in nanoseconds)`

This deadline is then checked against in the ArrowOptions contract:

```
if (currentNanoSec > openOptionParams.orderDeadline) {
            revert ExpiredOrderData(currentNanoSec, openOptionParams.order
Deadline);
        }
```

Because this parameter (orderDeadline) is not included in the signature, a different `orderDeadline` can be specified when opening a position through the router so that it passes the check `if (currentNanoSec > openOptionParams.orderDeadline)`, but the order params will be expired.

Recommendation

Add the following validation, to make sure the deadline is the actual deadline signed by the user.

```
+ uint256 currentNanoSec = getNanoSecFromSec(block.timestamp);
+ require(block.currentNanoSec < params.orderDeadline, "expired signature"
);

keccak256(
 abi.encode(
    keccak256(
 abi.encodePacked(
    'Param(bool longFlag,string ticker,uint256 expiration,uint256 strike1,
uint256 strike2,string decimalStrike,uint256 contractType,uint256 quantity
,uint256 thresholdPrice,uint256 signatureTimestamp)'
)
    ),
    params.longFlag,
    keccak256(abi.encodePacked(params.ticker)),
    params.expiration,
    params.strike[0],
    params.strike[1],
    keccak256(abi.encodePacked(params.decimalStrike)),
    params.contractType,
    params.quantity,
    params.thresholdPrice,
    params.signatureTimestamp
+   params.orderDeadline
                    )
                )
```

Resolution

Fixed

[AU-H2] Position Price signatures can be replayed.

Severity	Category	Status
HIGH	Signature	Resolved

Description

Position Price signatures can be replayed as soon as Arrow goes multichain, a fork deploys in the same chain, or there is a new version of the contract.

After deploying in Avalanche, Arrow will most likely also deploy on Arbitrum. The signature verification process can be seen in the following function:

```
  function verifyPositionPrice(OpenOptionParams calldata params) external
view returns (bytes32) {
        bytes32 message = keccak256(
            abi.encodePacked(
                keccak256(abi.encodePacked("Param(string ticker,uint256 ex
piration,uint256 strike1,uint256 strike2,uint256 contractType,uint256 quan
tity,uint256 optionPrice,uint256 currentTimestamp,uint256 deadline)")),
                params.ticker,
                params.expiration,
                params.strike[0],
                params.strike[1],
                params.contractType,
                params.quantity,
                params.optionPrice,
                params.positionParamsTimestamp,
                params.positionParamsDeadline
            )
        );
        bytes32 prefixedHash = message.toEthSignedMessageHash();
        // Verify marketmaker address
        address checkAddress = prefixedHash.recover(params.positionParamsS
ignature);
```

The issue here is that there is no domain separator. Allowing for the signature to be recovered in Arbitrum, other contracts in the same chain that use the same signature verification (forks), or even the same contract if upgraded.

Recommendation

Include a domain separator to account for all the cases on top that includes the `block.chain`, `address(this)`, a name and the `version`. It can also be called `EIP712Domain` for references.

Resolution

Fixed

[APM-H1] Users can “permanently freeze” market maker payout on short settlements.

Severity	Category	Status
HIGH	Protocol Loss	Resolved

Description

Currently, when settling an option, if it is a short, the user must send the `ownerPayoff` via `transferfrom` to the market maker.

The problem is the way on how the funds are sent as it requires an approval from the user.

```
  } else {
      // short position
      // Transfer payoff from owner to MarketMaker
        UserFundsManager(
            AbstractArrowRegistry(router.getRegistryAddress()).userFundsMa
nagerAddress()
                ).routeUserFunds('USDC', _owner, marketMakerAddressForFund
, ownerPayoff);
                // reduce locked Collateral
                _reduceCollateralForShortPosition(_optionData, _owner, opt
ionQuantity);
            }
            // Event
```

Therefore, a malicious user can simply not approve their funds to the market maker and the expired option will never be able to be settled.

This does not require a malicious user even, if the user is simply no active, either the payout will be massively delayed for the market maker, or it will never be sent.

Recommendation

Re-design the way on how payouts are handled when a position is short, and the user has to send funds to the market maker. Ideally you would like to directly take the funds from the user from some previously locked collateral that they sent before settlement.

Resolution

Fixed

[APM-H2] Malicious market makers can steal the options premium from users.

Severity	Category	Status
HIGH	User Loss	Acknowledged

Description

Market makers are currently expected to send the options cost when closing a long position. This is done via a transfer from the approval system. The market maker can decide to act maliciously and not return the premium to the user, by not approving the premium when closing a long position.

```
underlier.safeTransferFrom(source, destination, amount);
```

Recommendation

Do add a requirement so that market makers must also deposit collateral and lock it so they can't withdraw more than what is currently available in open positions that correspond to the market maker's address.

Resolution

Acknowledged. Even if they are third parties, we will keep them as a trusted role. Although on the future, this might change by forcing market makers to lock up collateral.

[APD-H1] Options can be settled with 0 as the settlement price.

Severity	Category	Status
HIGH	User Loss	Resolved

Description

Currently, the ideal or intended process on how an option gets settled, would be to initially call `initSettlement()` which is open for anyone to call, get the settlement price from gmx vault, and after call `settle()` which fetches the previously stored settlement price.

Unfortunately, there is no requirement for this to happen and `settle()` can be called directly by anyone without previously getting the settlement price, for that specific ticker. This will cause the settlement price to be 0, given that the mapping `settlementPrice[hashValue];` will not be updated.

Recommendation

Add the following check to the `settle()` function so that you can't settle without having a valid settlement price:

```
+ require(optionChainsSettleState[hashValue] == true, "price not fetched y
et");
```

Resolution

Fixed

[APD-H2] Options can be closed after expiration.

Severity	Category	Status
HIGH	User Loss	Resolved

Description

Currently, when closing a position, there is no requirement whatsoever to check that the expiration date of that option has not reached, allowing for that position to be closed even after the option has reached the expiration date.

Once options expire, they are no longer tradable, and you cannot close or exercise them. The expiration date is the last day on which an options contract is valid.

Recommendation

```
+ uint256 currentNanoSec = getNanoSecFromSec(block.timestamp);
+ require(block.currentNanoSec < closeOptionParams.expiration, "expired op
tion");
```

Add the following check when closing the position so that positions from expired options can't be "closed" because they should be settled.

Resolution

Fixed

[APD-H3] Users that open a position for the second time will get their initial position burned.

Severity	Category	Status
HIGH	User Loss	Resolved

Description

When opening a position for the first time, the function `openPositionDelegate()` is in charge to check if the option in fact exists already. If it does not exist, it creates and mints the option, if it exists it just continues.

In the case the option already exists and the user already has minted the same position previously, the following if statement will be reached:

```
 if (_id != 0) {

                uint256 _quantity = options.balanceOf(_openOptionParams.se
nderAddress, _id);
                if (_quantity != 0) {

                    if (openOptionParams.quantity > _quantity) {

                        openOptionParams.quantity = openOptionParams.quant
ity - _quantity;

                        options.burn(openOptionParams.senderAddress, _id,
_quantity);
                        if (options.totalSupply(_id) == 0) {
                            options.deleteId(_hashValue);
                        }
```

This basically means that when the user opens the same position twice (the quantity of options can be different) the previous position or option purchased, will be burned.

Recommendation

Do not burn the options from a previous position, rather just mint the new quantity on top.

Resolution

Fixed

[AR-H1] No requirement for the user to forward the gas to execute the transaction will incur losses for the router manager.

Severity	Category	Status
HIGH	Protocol Loss	Resolved

Description

The architecture that Arrow proposes requires the user to transfer some native gas from the function `depositGasFee()` because all the positions are opened and closed from the router, and the only one able to call those functions is the `onlyRouterManager()` role.

As of now there is no explicit requirement so that the `senderAddress` of the order has actual sent gas to cover for the execution costs from opening and closing positions. Allowing them to not send the required gas and incur in losses for the router manager.

Recommendation

Add the following require statement when opening and closing positions:

Additionally, the`200000` is just a placeholder. Do not hardcode it, make it a variable that can be changed and pre-calculate an estimation of the execution cost of the function to accurately price the execution cost.

```
  address positionManagerAddress = AbstractArrowRegistry(registryAddress).
getPositionManagerAddress();
+ require(gasPaid[senderAddress] >= (200000 * openOptionParams.length), "n
ot enough gas");
        for (uint256 i = 0; i < openOptionParams.length; ++i) {
            if (gasleft() < 200000) {
                revert GasLimitError();
            }
        AbstractArrowPositionManager(positionManagerAddress).openPosition(
openOptionParams[i]);
}
```

Resolution

Fixed

[APD-M1] Options can be created already in expiration and included in the `active` options list.

Severity	Category	Status
MEDIUM	DoS	Resolved

Description

Currently, there is no check or similar that makes sure when an order is being created, or a position is opened from the router that that option expiration is smaller than block.timestamp. This will add options as active while they should be expired.

Recommendation

Add the following check when creating the option/opening a position:

```
+ require(openOptionParams.expiration > block.timestamp; "creating an expi
red option");
```

If preferred, use an if revert instead of a require statement.

Resolution

Fixed

[APD-M2] Options can be transferred after expiration.

Severity	Category	Status
MEDIUM	User Loss	Resolved

Description

When creating a new option, as you can see in the following snippet, the `_expiration` is hashed with other params:

```
function createNewOption(
        string calldata _ticker,
        uint256 _expiration,
        uint256[2] calldata _strike,
        string calldata _decimalStrike,
        uint256 _contractType,
        address _marketMaker
    ) public override onlyPositionManager returns (uint256) {
        latestId++;
        bytes32 hashValue = keccak256( /
            abi.encodePacked(
                _longFlag,
                _ticker,
                _expiration,
                _decimalStrike,
                _contractType,
                _marketMaker
            )
        );
```

The options are ERC1155 tokens in this contract, all of them have different properties, one of them is the expiration date. A characteristic of options is that once they are expired, they must not be transferable. Unfortunately, there is no requirement in the contract that prevents to transfer each option after expiration.

Recommendation

Store the expiration date of every option and check against it in the transfer and transfer from function, so that it reverts on expirity:

```
  function safeTransferFrom(
        address from,
        address to,
        uint256 id,
        uint256 amount,
        bytes memory data
    ) public override onlyPositionManagerOrOptions(from) {
        if ((balanceOf(to, id) == 0) && (amount != 0)) _ownedTokens[to].pu
sh(id);

+     uint256 currentNanoSec = getNanoSecFromSec(block.timestamp);
+     if(currentNanoSec > _expiration){
+      revert ExpiredOrder();
+      }

        super._safeTransferFrom(from, to, id, amount, data);
        if (balanceOf(from, id) == 0) _removeId(from, id);
```

Resolution

Fixed

[UFM-M1] The `depositCollateral()` function does not handle the edge-case where ETH is sent but another ticker is specified.

Severity	Category	Status
MEDIUM	User Loss	Resolved

Description

In the `depositCollateral()` function, you can deposit your collateral by specifying the token/crypto you want to send via its ticker. There are several if statements that try to account for all the scenarios, as it does not behave the same if you send USDC as if you send the Native crypto, which in this case is AVAX.

The problem relies when someone specifies any token other than AVAX, but they are still allowed to send `msg.value`. If that would be the case, such `msg.value` would be lost and unaccounted for.

Recommendation

Add a requirement so that if the specified ticker is not ETH msg.value should be 0. An example:

```
+        if(keccak256(abi.encodePacked(ticker)) != keccak256(abi.encodePac
ked('AVAX'))){
+            require(msg.value == 0; "unsupported native value");
+         }
```

Resolution

Fixed

[UFM-M2] User collateral is underestimated when withdrawing.

Severity	Category	Status
MEDIUM	User Loss	Acknowledged

Description

The current architecture of the User Funds Manager contract makes a distinction between tokens and the native coin of the deployed chain, in this case AVAX. The collateral when sending tokens is tracked as the funds sent `totalDepositedAmounts[hashValue] += amount;` through the `depositCollateral()` function solely.

When the ticker is set to AVAX it not only tracks the funds from `depositCollateral()` as collateral, but also the fas paid from the `depositGasFee()` function.

In most of the calculations this rule is held true, except in the `withdrawCollateral()` function where `gasPaid[transactionSender]` is not accounted as collateral, miscalculating the total amount the user can withdraw.

Recommendation

Account for the `gasPaid[transactionSender]` also while withdrawing so that the user can completely withdraw their collateral.

Resolution

Acknowledged, the gas paid will not be accounted for when withdrawing.

[GLOBAL-M1] `address.call{value:x}()` should be used instead of `payable.transfer()` across the codebase

Severity	Category	Status
MEDIUM	User Loss	Resolved

Description

Several function from the codebase use .transfer() instead of .call() like `redeemGasFee()`

Using Solidity's transfer() function has some notable shortcomings when the withdrawer is a smart contract, which can render ETH deposits impossible to withdraw. Specifically, the withdrawal will inevitably fail when:

- The withdrawer is a multi-sig wallet (some multisig wallets use more 2300 on their fallback function)
- The withdrawer smart contract implements a payable fallback function which uses more than 2300 gas units.
- The withdrawer smart contract implements a payable fallback function which needs less than 2300 gas units but is called through a proxy that raises the call's gas usage above 2300.

Recommendation

Do use `address.call{value:x}()` instead of `transfer()`, always taking into consideration possible re-entrancy issues.

Resolution

Fixed

[GLOBAL-M2] Overall incompatibility for multichain deployment

Severity	Category	Status
MEDIUM	User Loss	Resolved

Description

As specified by the team, most likely in the future, they might expand to Arbitrum other than avalanche.

There are several issues right now, that will cause problems when deploying on arbitrum:

- The `address public stablecoinAddress = 0xB97EF9Ef8734C71904D8002F8b6Bc66Dd9c48a6E;` is hardcoded to USDC in avalanche. This address will not be the same in arbitrum
- Most of the if cases that require to fecth tickers, reference AVAX as the native crypto: `if (keccak256(abi.encodePacked(ticker)) == keccak256(abi.encodePacked('AVAX'))) {`
- The Natspec references AVAX directly: `if underlier is not avax token, check if lockable amount is greater than lock amount.`

Recommendation

- Pass the stablecoin address in the constructor instead of hardcoding it
- Chage the if cases from AVAX to `NATIVE`. And update the scripts that are in charge of adding tokens from its ticker.
- Grep for all the `avax` words in the codebase and update them to Native (referencing native token)

Resolution

Fixed

[GLOBAL-M3] Corrupted upgradeability pattern.

Severity	Category	Status
MEDIUM	Architectural Error	Resolved

Description

The contracts `ArrowEvents`, `AbstractArrowOptions`, `AbstractArrowPositionManager` , `AbstractArrowRegistry`, `ArrowRouter` , `AbstractUserFundsManager`, `AbstractArrowPositionManager` do not have gap storage implemented.

Thus, adding new storage variables to any of these inherited contracts can potentially overwrite the beginning of the storage layout of the child contract. causing critical misbehaviors in the system.

Recommendation

Consider defining an appropriate storage gap in each upgradeable parent contract at the end of all the storage variable definitions as follows:

`uint256[50] __gap; // gap to reserve storage in the contract for future variable additions`

You also can check the following report for reference:

https://blog.openzeppelin.com/notional-audit/

Resolution

Fixed

[GLOBAL-M4] Logic contracts can be destroyed.

Severity	Category	Status
MEDIUM	Architectural Error	Partially Resolved

Description

In the contracts implement Openzeppelin's UUPS model, uninitialized implementation contract can be taken over by an attacker with initialize function, it's recommended to invoke the `_disableInitializers` function in the constructor to prevent the implementation contract from being used by the attacker. However, all the contracts which implements OwnablePausableUpgradeable do not call `_disableInitializers` in the constructors

Recommendation

Add the following in all the implementation contracts that do use Openzeppelin's UUPS model. Some of them are:

- ArrowEvents
- ArrowOptions
- ArrowRegistry
- ArrowFundsManager
- ArrowPositionManger
- ArrowRouter
- AERC1155

```
/// @custom:oz-upgrades-unsafe-allow constructor
    constructor() {
        _disableInitializers();
    }
```

For more context, read: https://forum.openzeppelin.com/t/uupsupgradeable-vulnerability-post-mortem/15680

Resolution

Partially Fixed. `disableInitializers()` has not been implemented, rather the implementation init function is being called directly by arrow.

[APM-M1] Push over Pull system might cause position to be un-closable and market makers losing funds.

Severity	Category	Status
MEDIUM	Architectural Error	Partially Resolved

Description

The overall architecture of routing funds when opening/closing and settling positions/options, uses a push method where it directly transfers the required funds to each party. It could be the fee address, the owner of the position, or the market maker, the case is that in all scenarios those funds are pushed via a direct erc20 transfer to the respective address.

When routing funds, the only token that is used is USDC. Which, if you check on the arbiscan explorer: https://arbiscan.io/token/0xaf88d065e77c8cc2239327c5edb3a432268e5831 USDC does not allow transfers from or to blacklisted addresses.

If a user gets blacklisted while the option is still not expired, the settlements and the function to close position will revert, creating a loss for the market maker in case of shorts.

Recommendation

Do use a pull over push system where instead of directly transferring funds, add them to the collateral mappings so that any party is able to withdraw by themselves.

Resolution

Partially Fixed. The architecture has been fixed in most of the codebase, but not all.

[AR-M1] Users can grief the router manager by creating un-executable positions.

Severity	Category	Status
MEDIUM	Grieving	Acknowledged

Description

As of now, users can create positions with parameters that will revert, like passing a market maker that does not have a `marketMakerAddressForFund` related to it, or using any other param that will cause a revert on execution:

```
if (marketMakerAddressForFund == address(0)) {
            revert MarketMakerNotRegistered(openOptionParams.marketMaker);
        }
```

This will cause the transaction to revert every time it is executed by the router manager.

Recommendation

Adopt a similar system to what GMX has. Simulate the transaction (in this case opening and closing orders) before executing them with the router manager to make sure the transaction will not revert, if the simulation reverted, do not execute the position.

Resolution

Acknowledged. While there is not exact system that simulates transactions, some parameters like the market maker will be checked from an API.

[AU-L1] Incorrect pricing for settlements.

Severity	Category	Status
LOW	Protocol Loss	Acknowledged

Description

Currently, the price of the assets is fetched via the following call to a GMX vault:

```
 uint256 underlierPrice = (IVault(gmxVaultAddress).getMaxPrice(underlier)
+ IVault(gmxVaultAddress).getMinPrice(underlier)) / 2;
```

What Arrow is doing is fetching both the min and max prices from the vault and dividing them between 2. This will get the average between both prices. This is incorrectly done given that both, min, and max prices have different use cases and scenarios where to call each other.

`getMaxPrice()` gets the higher price, while `getMinPrice()` gets the lower price. The idea here, is to round against the user so they can't leverage rounding errors to their favor.

When u are buying you want to be buying at the highest price if it is long `getMaxPrice()` and selling it at the lower price `getMinPrice()`

When it is short buy at the lowest price `getMinPrice()` and sell at the highest price `getMaxPrice()`

Recommendation

Add a boolean to specify which one to call in each situation:

```
function getUnderlierPrice(
        address _routerAddress,
        string memory _ticker,
+     bool _maxPrice
    ) external view returns (uint256) {

+ if (_maxPrice){
+ uint256 underlierPrice = (IVault(gmxVaultAddress).getMaxPrice(underlier)
+ }else{
+  uint256 underlierPrice = IVault(gmxVaultAddress).getMinPrice(underlier)
);
+  }
```

Resolution

Acknowledged. There should not be different high or low prices for settlement. We've discussed our implementation with market makers as well and they said it was acceptable.

[UFM-L1] Check that the token which is being deposited is on the `tickerToAssetAddress` mapping.

Severity	Category	Status
LOW	Input Validation	Resolved

Description

On the `depositCollateral()`, you can specify the ticker of the asset you want to deposit and the amount.

If that asset is not the Native currency and it is not in the `tickerToAssetAddress`, the mapping will return `address(0)` and try to transfer as if the `underlier` was `address(0)` after.

```
  else {
            if (amount == 0) {
                return;
            }
            // if not avax token, total deposited Amount would be increase
d by amount
            // transfer underlier token from msg sender to UserFundsManage
r contract
            totalDepositedAmounts[hashValue] += amount;
            underlier.safeTransferFrom(msg.sender, address(this), amount);
        }
```

Recommendation

Do not allow users to try and deposit tokens that are not in the “whitelist”.

Add the following check:

```
 if (keccak256(abi.encodePacked(ticker)) == keccak256(abi.encodePacked('US
DC'))) {
            underlier = IERC20Metadata(
                AbstractArrowRegistry(router.getRegistryAddress()).stablec
oinAddress()
            );
        } else {
            underlier = IERC20Metadata(
                AbstractArrowRegistry(router.getRegistryAddress()).tickerT
oAssetAddress(ticker)
            );

+        require(underlier != address(0); "not supported token");
        }
```

Resolution

Fixed

[APM-L1] Hardcoding `loopCount` to 50 will be inaccurate.

Severity	Category	Status
LOW	Composability	Resolved

Description

When checking upKeep, they do loop through maximum 50 active options:

```
uint256 loopCount = 50;
```

There is no room to change this number to be more or less to be more accurate in the future. Gas costs of certain opcodes might change making it possible to add more loops or less depending if the gas increases or decreases.

Recommendation

Make `loopCount` a state variable and add a setter function to change its value if needed

```
- uint256 loopCount = 50;
+ uint256 _loopCount = loopCount;
```

Resolution

Fixed

[AE-L1] The `emitOptionTransfer` event has no access control to be emitted.

Severity	Category	Status
LOW	Access Control	Resolved

Description

On the ArrowEvents contract, the `emitOptionTransfer` event has no access control to be emitted.

```
function emitOptionTransfer(
        OptionDataType calldata params,
        uint256 id,
        address from,
        address to,
        uint256 amount
    ) external override {
        emit OptionTransfer(
            params.ticker,
            params.expiration,
            params.strike,
            params.decimalStrike,
            params.contractType,
            params.marketMaker,
            id,
            from,
            to,
            amount
        );
    }
```

This will allow anyone to emit the event as many times as need faking the actual indexing of the event.

Recommendation

Add the access control

```
function emitOptionTransfer(
        OptionDataType calldata params,
        uint256 id,
        address from,
        address to,
        uint256 amount
-   ) external override {
+ ) external override onlyPositionManager {
```

Resolution

Fixed

[GLOBAL-L1] Unused imports across the codebase.

Severity	Category	Status
LOW	Un-used imports	Resolved

Description

This is a list of unused imports across the codebase.

- ArrowRouter.sol: `import {UserFundsManager} from './UserFundsManager.sol';`
- ArrowRegistry.sol: `import {IArrowRouter} from './abstracts/IArrowRouter.sol';`
- ArrowPositionManager: `import {IVault} from './oracle/IVault.sol';`
- The error: SenderNotCorrectPositionManager is unused in AbstractArrowRegistry

Recommendation

Delete the unused imports, errors.

Resolution

Fixed

[GLOBAL-L2] Protocol does not work with transfer-tax tokens and/or erc777s.

Severity	Category	Status
LOW	Architectural Error	Resolved

Description

The whole architecture has several spots which will not work with transfer-tax tokens and erc777 tokens, this will break the whole system given that it allows for re-entrancies and the balances would be tracked incorrectly.

Recommendation

Given that supporting them would need a major refactoring on the code, my recommendation is to simply not use such tokens.

Resolution

Fixed

[AO-L1] Superfluous `address(0)` check when verifying the option Id.

Severity	Category	Status
LOW	Input Validation	Resolved

Description

The verification of the option Id is used twice across the codebase, when opening and closing positions.

At the start of the `verifyId()` function it checks that the sender address is not 0: `if (openOptionParams.senderAddress == address(0))`

Though, this is redundant as it has already been checked before in the `_checkAddressLogic(openOptionParams.senderAddress);` function when opening and closing positions

Recommendation

Remove the second check:

```
function verifyId(
        OpenOptionParams calldata openOptionParams
    ) public override onlyPositionManager returns (uint256 id) {
-        if (openOptionParams.senderAddress == address(0)) {
-            revert SenderIsZeroAddress();
        }
        uint256 currentNanoSec = ArrowUtils.getNanoSecFromSec(block.timest
amp);
```

Resolution

Fixed

[AERC1155-L1] Redundant inheritance in AERC1155.

Severity	Category	Status
LOW	Architectural Error	Resolved

Description

In the AERC1155 contract, when inheriting, ERC1155 is being inherited several times making the inheritance redundant and shadowing function:

```
abstract contract AERC1155 is ERC1155, ERC1155Burnable, Ownable,
ERC1155Supply {
```

Recommendation

Delete the ERC1155 import as it is already in ERC1155Supply:

```
- abstract contract AERC1155 is ERC1155, ERC1155Burnable, Ownable, ERC1155
Supply {
+ abstract contract AERC1155 is ERC1155Burnable, ERC1155Supply, Ownable {
```

Resolution

Fixed

[AR-L1] Hardcoding the minimum gasLeft brakes composability.

Severity	Category	Status
LOW	Composability	Resolved

Description

In the router, calling `openPosition()` allows to batch create positions. Currently there is a check that if `gasleft() < 200000`, it will revert:

```
for (uint256 i = 0; i < openOptionParams.length; ++i) {
            if (gasleft() < 200000) {
                revert GasLimitError();
            }
```

Allow the `200000` to be a variable and being able to customize it if needed on the future, otherwise `200000` might be inaccurate.

Recommendation

Add the state variable `minGasLeft` and a setter function with access control to update it on the future

```
 for (uint256 i = 0; i < openOptionParams.length; ++i) {
-           if (gasleft() < 200000) {
+        if (gasleft() < minGasLeft) {
                revert GasLimitError();
            }
```

Resolution

Fixed

[AR-L2] Use a 2-step ownership transfer.

Severity	Category	Status
LOW	Ownership	Resolved

Description

Arrow Markets uses a single-step access control transfer pattern in the router contract. This means that if the current owner account transfers ownership with an incorrect address, then this owner role will be lost forever along with all the functionality that depends on it.

Recommendation

Follow the pattern from OpenZeppelin's Ownable2Step and implement a two-step transfer pattern for the action.

Resolution

Fixed

[APD-L1] Superfluous **`optionPriceWithFee`** calculation when opening and closing positions.

Severity	Category	Status
LOW	Input Validation	Resolved

Description

When opening and closing positions in the ArrowPositionDelegate contract, there is a calculation that is being done to calculate the `optionPriceWithFee` though it is not used anywhere in the contracts:

```
{
            uint256 optionPriceWithFee;
            if (closeOptionParams.longFlag) {
                // long position
                optionPriceWithFee =
                    closeOptionParams.optionPrice -
                    registry.computeOrderFee(
                        closeOptionParams.optionPrice,
                        closeOptionParams.longFlag
                    );
            } else {
                // short position
                optionPriceWithFee = closeOptionParams.optionPrice;
            }
        }
```

Recommendation

Remove the code snippet attached before opening and closing positions.

Resolution

Fixed

[APD-L2] Superfluos `Id` fetching.

Severity	Category	Status
LOW	Input Validation	Resolved

Description

The following code when opening a position in Arrow Position Delegate is redundant as the `id` is fetched and `hashed on top, as can be seen here:` uint256 id = options.verifyId(openOptionParams);`

Therefore, making all the extra code redundant:

```
            // check if sender has open options
            OpenOptionParams memory _openOptionParams = abi.decode(packedD
ata, (OpenOptionParams));

            _openOptionParams.longFlag = !openOptionParams.longFlag;

            bytes32 _hashValue = keccak256(
                abi.encodePacked(
                    _openOptionParams.longFlag,
                    _openOptionParams.ticker,
                    _openOptionParams.expiration,
                    _openOptionParams.decimalStrike,
                    _openOptionParams.contractType,
                    _openOptionParams.marketMaker
                )
            );
            uint256 _id = options.ids(_hashValue);
```

Recommendation

Remove the redundant code:

```
-     // check if sender has open options
-              OpenOptionParams memory _openOptionParams = abi.decode(pack
edData, (OpenOptionParams));

-              _openOptionParams.longFlag = !openOptionParams.longFlag;

-              bytes32 _hashValue = keccak256(
-                  abi.encodePacked(
-                      _openOptionParams.longFlag,
-                      _openOptionParams.ticker,
-                      _openOptionParams.expiration,
-                      _openOptionParams.decimalStrike,
-                     _openOptionParams.contractType,
-                      _openOptionParams.marketMaker
-                  )
-             );
-             uint256 _id = options.ids(_hashValue);
```

Resolution

Fixed

[APD-L3] Missing check: `_checkCollateralBalance()` before reducing lock.

Severity	Category	Status
LOW	Input Validation	Resolved

Description

Currently, when closing a position, the lock of the previously deposited collateral funds has to be released. There are 2 paths for it, when a closing position is long, and when a closing position is short. Currently, when the closing position is long, the check:

```
 _checkCollateralBalance(
                            'USDC',
                            closeOptionParams.senderAddress,
                            requiredAmount
                        );
```

is missing, while in the short case, it is present.

Recommendation

Add the following check in the case the closing position is a long.

```
 if (requiredAmount == 0) {
                            return;
                        }
+                     _checkCollateralBalance('USDC',closeOptionParams.sen
derAddress, requiredAmount);
                        // reduce locked Collateral
                        UserFundsManager(registry.userFundsManagerAddress(
))
                            .reduceLockDepositedAmount(
                                closeOptionParams.ticker,
                                closeOptionParams.senderAddress,
                                requiredAmount
                            );
```

Resolution

Fixed

[AO-INF1] Incorrect comment section

Severity	Category	Status
INF	Natspec	Resolved

Description

On the Arrow Options contract, at the beginning, it is specified as a comment section:

```
/*********************
    *     CONSTRUCTOR      *
 *********************/
```

But there is no constructor in the file.

Recommendation

Delete the comment section.

Resolution

Fixed

[GLOBAL-INF] Gas improvements across the codebase

Severity	Category	Status
INF	Gas	Resolved

Issues

This is a list of gas improvements that can be done to the code:

- For loops are declared: `for (uint256 i = 0; i < openOptionParams.length; ++i) {`

Do not initialize `uint256 i = 0` to 0

```
- for (uint256 i = 0; i < openOptionParams.length; ++i) {
+ for (uint256 i ; i < openOptionParams.length; ++i) {`
```

- On the `_calculateDiffPrice` function in the Arrow Position Manager and the Arrow Position Delegate contracts initializes the uint with a 0. This will cost more gas and it is redundant as by default a uint without initialization is already 0.

Change it to:

```
- uint256 diffPrice = 0;
+ uint256 diffPrice;
```

Resolution

Fixed

DISCLAIMER

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Marc Weiss to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Blockchain technology and cryptographic assets present a high level of ongoing risk.

My position is that each company and individual are responsible for their own due diligence and continuous security. My goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze. Therefore, I do not guarantee the explicit security of the audited smart contract, regardless of the verdict.