Aelin DAO - AELIP-41
Security Assessment Findings Report

Date: April 21, 2023
Project: AAE
Version 1.0

Contents

1 Confidentiality statement

This document is the exclusive property of Aelin DAO and Sub7 Security. This document contains proprietary and confidential information. Duplication, redistribution, or use, in whole or in part, in any form, requires consent of both Aelin DAO and Sub7 Security.

2 Disclaimer

A smart contract security audit is considered a snapshot in time. The findings and recommendations reflect the information gathered during the assessment and not any changes or modifications made outside of that period.

Time-limited engagements do not allow for a full evaluation of all security controls. Sub7 Security prioritized the assessment to identify the weakest security controls an attacker would exploit. Sub7 Security recommends conducting similar assessments on an annual basis by internal or third-party assessors to ensure the continued success of the controls

3 About Sub7

Sub7 is a Web3 Security Agency, offering Smart Contract Auditing Services for blockchain-based projects in the DeFi, Web3 and Metaverse space.

Learn more about us at https://sub7.xyz

4 Project Overview

Aelin is a permissionless multi-chain protocol for capital raises and OTC deals. It allows anyone to raise capital for their project or an OTC deal and to seek out investors.

The Aelin protocol is designed to enable protocols and sponsors to raise money from their communities in two primary ways:

1. *Aelin Direct Deals* are an excellent option for new and established protocols looking to raise capital from investors at pre-established deal terms. Aelin Deals allow for customizable features, such as a defined vesting period and a vesting cliff. Additionally, these deals can also include NFT-gated features, which can be a great way to restrict access to particular NFT holders. Overall, Aelin Deals are a great way to raise capital for new protocols.
2. *Aelin Pools* are best suited for protocols that don't have set deal terms yet, but are gauging investor interest. This type of pool is ideal for sponsors using Aelin to source a future deal, as it allows them to collect data and feedback from investors without committing to anything. Aelin Pools are most similar to SPACs in terms of their purpose and structure.

Aelin is proud to offer many custom features to help protocols and sponsors raise capital.

1. *NFT Gated Pools* - Protocols can decide to gate off their token raise to holders of specific NFT collections. This allows protocols and DAOs to target specific groups of investors, which can improve the process of building and growing their community. Protocols can set maximums purchase amounts for each NFT holder.
2. *Custom Allow List* (100k addresses+) - Protocols may want to provide rewards to a large number of addresses with different allowance amounts to access their upfront Aelin deal. Protocols or large holders can use an off-chain script to generate a CSV file with several hundred thousand addresses and custom amounts based on each wallet's activity level onchain or any logic the protocol desires.
3. *Customizable Vesting Options* - Protocols that use Aelin can customize their vesting options (e.g. cliff, vesting time, etc.) to ensure that their project's objectives are aligned with their investors' incentives.

4. *Predetermined Deal Terms* - Aelin provides protocols with the ability to set a specific, fixed exchange rate for investors, without the worry of fluctuating market conditions affecting the terms of the deal. This makes Aelin more like a large on-chain OTC transaction, which is preferable for protocols that are targeting specific valuations.

5 Executive Summary

Sub7 Security has been engaged to what is formally referred to as a Security Audit of Solidity Smart Contracts, a combination of automated and manual assessments in search for vulnerabilities, bugs, unintended outputs, among others inside deployed Smart Contracts.

The goal of such a Security Audit is to assess project code (with any associated specification, and documentation) and provide our clients with a report of potential security-related issues that should be addressed to improve security posture, decrease attack surface and mitigate risk.

2 (Two) Security Researchers/Consultants were engaged in this activity.

5.1 Scope

Assessment: https://github.com/AelinXYZ/aelin/tree/53710152d3746cbfc5337e88a3f01694d5b26999/contracts

Verification: https://github.com/AelinXYZ/aelin/commit/0d94add991cef4cf85a02c5cbfd83c7148b3e51d/contracts

5.2 Timeline

Assessment: 27 March 2023 to 6 April 2023

Verification: 18 April 2023

5.3 Summary of Findings Identified

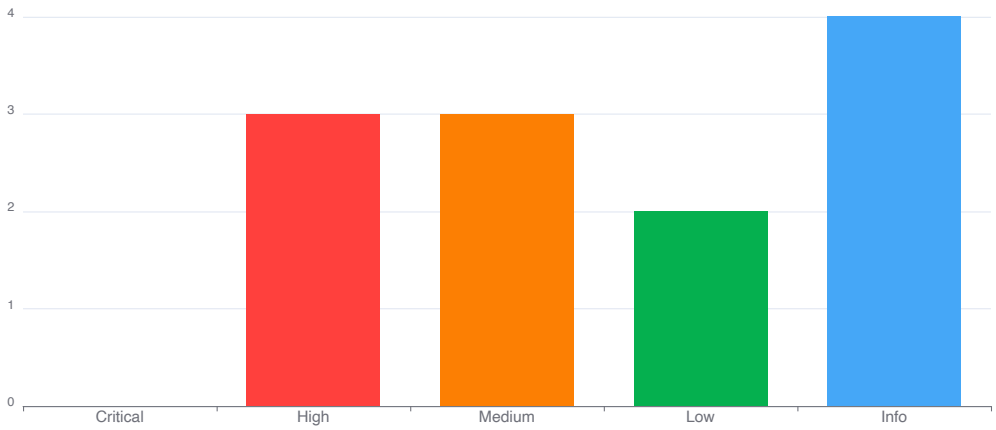

Figure 1: Executive Summary

 # 1 High Malicious purchaser can bypass validation checks – ***Fixed***

 # 2 High Sponsor's fee can be forced trapped in contract permanently – ***Fixed***

 # 3 High Wrong handling of decimals for underlying pool tokens that are not 18 decimals – ***Fixed***

 # 4 Medium Multiple flaws related to ERC1155 tokens – ***Fixed***

 # 5 Medium No support for NFTs collections with both standards (ERC1155 and ERC721) – ***Acknowledged***

 # 6 Medium Large Purchasers have advantage over smaller ones – ***Acknowledged***

 # 7 Low Update external dependency to latest version – ***Open***

 # 8 Low Multiple Unchecked Array Lengths – ***Acknowledged***

 # 9 Informational Update of solidity safe pragma – ***Open***

 # 10 Informational Prefer Solidity Custom Errors over require statements with strings – ***Acknowledged***

 # 11 Informational `setHolder` should emit an event – ***Fixed***

 # 12 Informational NatSpec docs are missing – ***Open***

5.4 Methodology

Sub7 Security follows a phased assessment approach that includes thorough application profiling, threat analysis, dynamic and manual testing. Security Auditors/Consultants utilize automated security tools, custom scripts, simulation apps, manual testing and validation techniques to scan for, enumerate and uncover findings with potential security risks.

As part of the process of reviewing solidity code, each contract is checked against lists of known smart contract vulnerabilities, which is crafted from various sources like SWC Registry , DeFi threat and previous audit reports.

The assessment included (but was not limited to) reviews on the following attack vectors:

Oracle Attacks | Flash Loan Attacks | Governance Attacks | Access Control Checks on Critical Function | Account Existence Check for low level calls | Arithmetic Over/Under Flows | Assert Violation | Authorization through tx.origin | Bad Source of Randomness | Block Timestamp manipulation | Bypass Contract Size Check | Code With No Effects | Delegatecall | Delegatecall to Untrusted Callee | DoS with (Unexpected) revert | DoS with Block Gas Limit | Logical Issues | Entropy Illusion | Function Selector Abuse | Floating Point and Numerical | Precision | Floating Pragma | Forcibly Sending Ether to a Contract | Function Default Visibility | Hash Collisions With Multiple Variable Length Arguments | Improper Array Deletion | Incorrect interface | Insufficient gas griefing | Unsafe Ownership Transfer | Loop through long arrays | Message call with hardcoded gas amount | Outdated Compiler Version | Precision Loss in Calculations | Price Manipulation | Hiding Malicious Code with External Contract | Public burn() function | Race Conditions / Front Running | Re-entrancy | Requirement Violation | Right-To-Left-Override control character (U+202E) | Shadowing State Variables | Short Address/Parameter Attack | Signature Malleability | Signature Replay Attacks | State Variable Default Visibility | Transaction Order Dependence | Typographical Error | Unchecked Call Return Value | Unencrypted Private Data On-Chain | Unexpected Ether balance | Uninitialized Storage Pointer | Unprotected Ether Withdrawal | Unprotected SELFDESTRUCT Instruction | Unprotected Upgrades | Unused Variable | Use of Deprecated Solidity Functions | Write to Arbitrary Storage Location | Wrong inheritance | Many more…

6 Findings and Risk Analysis

6.1 Malicious purchaser can bypass validation checks

Severity: High
Status: Fixed

Description

AelinPool.purchasePoolTokensWithNft is expected to allow anyone to become a purchaser with a qualified ERC721 NFT in the pool.

Although there are many validation checks on the elements in the `NftPurchaseList\[\]` array, such as the types of NFT collections, the desired purchase amount, the balances of the purchaser, etc., but one important check is missing: validating that the passed `\_nftPurchaseList\[i\].collectionAddress` is not the null address.

In this case, the null address will be required to be equal to the stored `collectionAddress` property for this collection in the `nftCollectionDetails` mapping and will indeed be equal because the default address-type value will be returned, which is again the null address.

All checks after that will also pass:

```
if (nftCollectionRules.purchaseAmount == 0) {
    maxPurchaseTokenAmount = \_purchaseTokenAmount;
}
```

The above check will pass because the “purchaseAmount” will have the default unsigned integer value of 0.

```
if (NftCheck.supports721(collectionAddress)) {
    \_blackListCheck721(collectionAddress, tokenIds);
}
if (NftCheck.supports1155(collectionAddress)) {
    \_eligibilityCheck1155(collectionAddress, tokenIds, nftCollectionRules);
}
```

The above two checks will also pass because the returned data from the low-level call in NftCheck._supportsInterface will have a size of 0 because the call was made to an EOA (the null address):

```
function \_supportsInterface(address account, bytes4 interfaceId) private view returns (
    bool) {
    bytes memory encodedParams = abi.encodeWithSelector(IERC165.supportsInterface.selector
        , interfaceId);
    (bool success, bytes memory result) = account.staticcall{gas: 30000}(encodedParams);
    if (result.length < 32) return false;
```

```
5         return success && abi.decode(result, (bool));
6     }
```

As a result, any user can purchase pool tokens, even if the pool configuration expects only owners of specific NFTs to be allowed to do so. In other words, this vulnerability allows user to make **arbitrary number** of purchases and without the required NFT. In fact, it is likely legitimate users who own the required NFTs will not be able to make their purchases with the `maxPurchaseTokenAmount` if this vulnerability is abused.

Location

https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/AelinPool.sol#L223

https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/AelinPool.sol#L236-L237

https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/AelinPool.sol#L249-L251

https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/libraries/NftCheck.sol#L23-L24

Recommendation

Check if `nftPurchaseList.collectionAddress` is address(0)

Client Comments

None

6.2 Sponsor's fee can be forced trapped in contract permanently

Severity: High
Status: Fixed

Description

`sponsorClaim()` can be made only once, and it can be called after purchase expiry. However, purchase expiry only denotes the stage after which purchasers has invested in pool tokens, not the stage after which deal tokens are accepted. `totalSponsorFeeAmount` is based on the amount of deal tokens that are accepted by Purchasers, i.e. purchasers agreeing and accepting with the deal terms made by Sponsor.

```
function sponsorClaim() external {
    require(block.timestamp >= purchaseExpiry, "still in purchase window");
    require(sponsorClaimed != true, "sponsor already claimed");
    require(totalSponsorFeeAmount > 0, "no sponsor fees");

    sponsorClaimed = true;
    aelinDeal.mintVestingToken(sponsor, totalSponsorFeeAmount);
}
```

If `sponsorClaim()` is called after purchase period ends and as long as at least one deal has been accepted (since we require `totalSponsorFeeAmount > 0`, then fees collected from future deal tokens that are accepted will be lost forever, as sponsor can only claim once.

In fact, since `sponsorClaim()` is a public function, an adversary can prevent sponsor from collecting any fees by accepting a deal right after purchase expiry (even one of a very small amount) and calling `sponsorClaim()`.

Location

https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/AelinPool.sol#L501-L508

Recommendation

Consider allowing `sponsorClaim()` to be called multiple times.

```
function sponsorClaim() external {
    require(block.timestamp >= purchaseExpiry, "still in purchase window");
-   require(sponsorClaimed != true, "sponsor already claimed");
    require(totalSponsorFeeAmount > 0, "no sponsor fees");

-   sponsorClaimed = true;
    aelinDeal.mintVestingToken(sponsor, totalSponsorFeeAmount);
+   totalSponsorFeeAmount = 0;
}
```

Client Comments

None

6.3 Wrong handling of decimals for underlying pool tokens that are not 18 decimals

Severity: High
Status: Fixed

Description

vestingDetails[] tokens are saved in their wrapped form, i.e. 18 decimals. We can see from L551, when vesting tokens are minted, the formatted pool token amount is used.

```
function \_mintDealTokens(address \_recipient, uint256 \_poolTokenAmount) internal {
    \_burn(\_recipient, \_poolTokenAmount);
    uint256 poolTokenDealFormatted = \_convertPoolToDeal(\_poolTokenAmount,
        purchaseTokenDecimals);
    uint256 aelinFeeAmt = (poolTokenDealFormatted \* AELIN\_FEE) / BASE;
    uint256 sponsorFeeAmt = (poolTokenDealFormatted \* sponsorFee) / BASE;

    totalSponsorFeeAmount += sponsorFeeAmt;

    aelinDeal.transferProtocolFee(aelinFeeAmt);
    aelinDeal.mintVestingToken(\_recipient, poolTokenDealFormatted - (sponsorFeeAmt +
        aelinFeeAmt)); // L551

    IERC20(purchaseToken).safeTransfer(holder, \_poolTokenAmount);
    emit AcceptDeal(\_recipient, address(aelinDeal), \_poolTokenAmount, sponsorFeeAmt,
        aelinFeeAmt);
}
```

When underlying tokens are claimed, they are retrieved based on the amount value in `vestingDetails \[\]`.

```
function claimableUnderlyingTokens(uint256 \_tokenId) public view returns (uint256) {
    VestingDetails memory schedule = vestingDetails\[\_tokenId\];
    uint256 precisionAdjustedUnderlyingClaimable;

    if (schedule.lastClaimedAt > 0) {
        uint256 maxTime = block.timestamp > vestingExpiry ? vestingExpiry : block.
            timestamp;
        uint256 minTime = schedule.lastClaimedAt > vestingCliffExpiry ? schedule.
            lastClaimedAt : vestingCliffExpiry;

        if (maxTime > vestingCliffExpiry && minTime <= vestingExpiry) {
            uint256 underlyingClaimable = (schedule.share \* (maxTime - minTime)) /
                vestingPeriod;

            // This could potentially be the case where the last user claims a slightly
                smaller amount if there is some precision loss
            // although it will generally never happen as solidity rounds down so there
                should always be a little bit left
            precisionAdjustedUnderlyingClaimable = underlyingClaimable >
                IERC20(underlyingDealToken).balanceOf(address(this))
                ? IERC20(underlyingDealToken).balanceOf(address(this))
                : underlyingClaimable;
        }
    }
    return precisionAdjustedUnderlyingClaimable;
}
```

Underlying deal tokens that are not 18 decimals will be problematic. When underlying deal tokens are claimed, the amount being claimed is going to be many times more than the correct amount, based on the number of differences in decimals between underlyingDealToken.decimals() and 18. Though, it is likely that this call will fail as contract does not have enough underlying deal tokens to transfer to claimant.

Note that `totalUnderlyingClaimed` is also updated based on this wrapped value. When we withdraw,

the same decimal issue computation is seen on L140.

```
function withdraw() external onlyHolder {
    uint256 withdrawAmount;
    if (!depositComplete && block.timestamp >= holderFundingExpiry) {
        withdrawAmount = IERC20(underlyingDealToken).balanceOf(address(this));
    } else {
        withdrawAmount =
            IERC20(underlyingDealToken).balanceOf(address(this)) -
            (underlyingDealTokenTotal - totalUnderlyingClaimed); // L140
    }
    IERC20(underlyingDealToken).safeTransfer(holder, withdrawAmount);
    emit WithdrawUnderlyingDealToken(underlyingDealToken, holder, withdrawAmount);
}
```

Location

https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/AelinPool.sol#L551
https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/AelinDeal.sol#L180
https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/AelinDeal.sol#L140

Recommendation

We should convert `claimableAmount` back to underlying deal token decimals using `underlyingPerDealExchangeRate` .

Client Comments

None

6.4 Multiple flaws related to ERC1155 tokens

Severity: Medium
Status: Fixed

Description

There are multiple issues related to ERC1155 tokens when using AelinPool.purchasePoolTokensWithNft to purchase pool tokens.

Firstly, the current implementation does not take into account the purchaser's balance of a specific ERC1155 collection token. AelinNftGating.purchaseDealTokensWithNft considers the purchaser's balance by adding `purchaseAmount` to the `maxPurchaseTokenAmount` for each instance (count) of each

specific token when `purchaseAmountPerToken` is set to true and the NFT token implements the ERC1155 standard. This logic is missing in AelinPool.purchasePoolTokensWithNft which may result in wrong `maxPurchaseTokenAmount` for some purchasers.

Secondly, there is a flaw with the `nftWalletUsedForPurchase`. Purchasers can get around the following check if their NFT collection is an ERC1155:

```
require(!nftWalletUsedForPurchase\[collectionAddress\]\[msg.sender\], "wallet already used for
 nft set");
```

The mapping is expected to restrict purchasers from making multiple purchases with the same NFT when `purchaseAmountPerToken` is false. However, this can be bypassed by transferring the allowed ERC1155 tokens to another account controlled by the same purchaser, which allows them to make another purchase for the same maximum amount allowed for the NFT token. The received pool tokens can then be transferred back to the original account.

Thirdly, there is an issue with accounting for the used amount of ERC1155 tokens when purchasing pool tokens. Currently, AelinPool._eligibilityCheck1155 only checks that the purchaser has more than `\_nftCollectionRules.minTokensEligible\[i\]` ERC1155 tokens, but it does not mark the used amount (count) of specific tokens (tokenId). This allows purchasers to buy more pool tokens than `nftCollectionRules.purchaseAmount` by simply calling AelinPool.purchasePoolTokensWithNft multiple times.

Location

https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/AelinPool.sol#L222-L262

Recommendation

For the first one consider using the `AelinNftGating` in AelinPool. For the second issue decide whether this check is useful at all or think of a way to not allow the above scenario to happen. For the third issue you can mark the used number of specific tokens of an ERC1155 collection and do not allow replay.

Client Comments

None

6.5 No support for NFTs collections with both standards (ERC1155 and ERC721)

Severity: Medium
Status: Acknowledged

Description

Some NFTs supports both the ERC1155 and ERC721 interface. For example, The Sandbox Game ASSETS token, a top 20 NFT with over 9000 ETH volume on Openseas, implements their asset token this way. Deployed ASSETS token contract code can be seen here.

The protocol currently assumes that an NFT must either be ERC1155 or ERC721. For example, if a dual asset token is added to the collection rules, with the first token being an ERC1155, `nftId\[collectionAddress\[tokenId\]` will be set to true. But in the `\_blackListCheck721()`, it requires that this value is `false`. Since a dual asset token is considered both ERC1155 and ERC721, it will break the protocol as purchases with NFT cannot be made.

Location

https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/AelinPool.sol#L253-L258
https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/AelinPool.sol#L283-L290

Recommendation

If the protocol intends to support such tokens, refactoring of the code is needed as current implementation assumes that NFT must either be ERC1155 or ERC721.

Client Comments

None

6.6 Large Purchasers have advantage over smaller ones

Severity: Medium
Status: Acknowledged

Description

Holders can choose to release underlying deal tokens at a certain percentage, with the remaining only claimable during the open redemption period. Purchasers knowing this, will be incentivesed to purchase more pool tokens than what they intend to spend.

For example, if conversion ratio is 50%, it means that a user with 1000 pool tokens will only be able to accept deal with 500 pool tokens during the proRataRedemption period. Hence, knowing this, if a user has the intention to spend 1000 pool tokens, the user is incentivsed to invest 2000 pool tokens initially,

and be able to accept the deal with how much they originally planned to despite the conversion ratio. Note that the remaining pool tokens can always be redeemed back for `purchaseToken`.

Because a Purchaser will not know of the terms of the deal beforehand, i.e. they do not know if conversion ratio will be set to 10% or 20% .... 100%, it is best for them to make a larger Pool token purchase initially.

A consequence of this is that it will quickly push up `purchaseTokenCap`, filling purchase token cap with many "fake purchases", it can limit and prevent legitimate users from participating in the deal. A large percentage of the `purchaseTokenCap` being "fake purchases" will also prevent Holder from successfully raising sufficient capital.

Location

https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/AelinPool.sol#L200-L206

Recommendation

Because there is an inherent economic incentive when conversion ratio is not 1:1, we can consider removing the possbility of setting a `purchaseTokenCap`, so that all Purchasers can invest regardeless.

Client Comments

None

6.7 Update external dependency to latest version

Severity: Low
Status: Open

Description

Update the versions of @openzeppelin/contracts to be the latest in package.json.

Location

According to package.json, "@openzeppelin/contracts" is currently set to ^4.5.0.

Recommendation

I also recommend double checking the versions of other dependencies as a precaution, as they may include important bug fixes.

Client Comments

None

6.8 Multiple Unchecked Array Lengths

Severity: Low
Status: Acknowledged

Description

Ethereum is a very resource-constrained environment. Prices per computational step are orders of magnitude higher than with centralized providers. Moreover, Ethereum miners impose a limit on the total number of Gas consumed in a block.

If `\_nftPurchaseList.length` is large enough, the function exceeds the block gas limit, and transactions calling it will never be confirmed.

This becomes a security issue, if an sponsor uses a huge `\_nftPurchaseList` or has a very big amount of NFTs looking to be valid for participation in a pool, possibly can cause the complete contract to be stalled.

Location

https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/libraries/AelinNftGating.sol#L112

Recommendation

Set limits to the amount of NFTs allowed to be used as part of a Pool

Client Comments

None

6.9 Update of solidity safe pragma

Severity: Informational
Status: Open

Description

All of the contracts use `0.8.6` version.

Location

This problem is present in all contracts.

Recommendation

Consider using a newer version of the compiler as the latest available version is `0.8.19` which has a lot of new features and optimizations.

Client Comments

None

6.10 Prefer Solidity Custom Errors over require statements with strings

Severity: Informational
Status: Acknowledged

Description

Using Solidity Custom Errors has the benefits of less gas spent in reverted transactions, better interoperability of the protocol as clients of it can catch the errors easily on-chain, as well as you can give descriptive names of the errors without having a bigger bytecode or transaction gas spending, which will result in a better UX as well.

Location

Require statements are used quite often through the main contracts.

Recommendation

Remove all require statements and use Custom Errors instead.

Client Comments

None

6.11 `setHolder` should emit an event

Severity: Informational
Status: Fixed

Description

In `AelinDeal.sol` the `setHolder}}event is missing in {{setHolder` method. Also consider declaring `acceptHolder` event and emitting it in `acceptHolder` method. The same is valid for `SetSponsor` event in `AelinPool.sol`.

Location

https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/AelinDeal.sol#L234

https://github.com/AelinXYZ/aelin/blob/53710152d3746cbfc5337e88a3f01694d5b26999/contracts/AelinDeal.sol#L239

Recommendation

State-changing methods should emit events so that off-chain monitoring can be implemented. Make sure to emit a proper event in each state-changing method to follow best practices.

Client Comments

None

6.12 NatSpec docs are missing

Severity: Informational
Status: Open

Description

In almost all methods the NatSpec documentation is incomplete as it is missing parameters and return variables in it. NatSpec documentation is essential for better understanding of the code by developers and auditors and is strongly recommended.

Location

All through the codebase.

Recommendation

Please refer to the NatSpec format and follow the guidelines outlined there.

Client Comments

None