Quantstamp

Ajna Finance (Governance)

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	Proposal System
Timeline	2023-04-24 through 2023-05-03
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	Ajna Protocol Whitepaper 03-10-2023 Ajna Protocol Master Spec README.md Ajna Protocol Whitepaper 03-24-2023
Source Code	• ajna-finance/ecosystem-coordination #245c6cb
Auditors	• Guillermo Escobero Auditing Engineer • Jonathan Mevs Auditing Engineer • Nikita Belenkov Auditing Engineer • Rabib Islam Auditing Engineer

Documentation quality	High
Test quality	High
Total Findings	13 Fixed: 11 Acknowledged: 2
High severity findings	3 Fixed: 3
Medium severity findings	0
Low severity findings	4 Fixed: 3 Acknowledged: 1
Undetermined severity findings	2 Fixed: 2
Informational findings	4 Fixed: 3 Acknowledged: 1

Summary of Findings

Fix review update: The Ajna Finance team provided a new commit containing fixes for the issues found. All the issues were addressed. The Extraordinary Funding Method was removed from the project, solving some of the initial issues found. Regarding AJN-4, it was acknowledged by the team. We still recommend including a warning for users in the documentation, as proposals containing a large amount of target addresses could fail due to running out of gas. The funds allocated for those failing proposals will be blocked.

Initial audit: Quantstamp audited Ajna Finance's governance smart contracts. Ajna Finance is marketed as a non-custodial, peer-to-peer, permissionless lending, borrowing, and trading system that requires no governance or external price feeds to function. Ajna Finance coordinates ecosystem growth in a decentralized voting system, where a fixed percentage of the treasury funds will be distributed in proposals submitted by the community in quarterly periods.

We have identified scenarios where the quarterly reserved funds can be locked in the smart contract (e.g. if a proposal execution fails or the rewards are not fully distributed). As per our classification, this would normally rank as "Medium" in severity, but given that blocked funds can be confusing for users, and the impact may be greater if the token price increases, we are increasing the severity to "High" (AJN-1, AJN-2, AJN-3). All issues and design recommendations are discussed in the *Findings* section of this document. Following that, recommendations about documentation and best practices are discussed. We strongly recommend addressing all the issues before deployment.

Regarding testing, all tests passed, and the project implements code coverage metrics, reaching a good coverage percentage (`93.33%` branch coverage). We recommend reaching `100%` in `StandardFunding.sol`. Although code coverage is high, some issues found show that some corner cases are not tested (AJN-1, AJN-2, AJN-3). We recommend covering these scenarios and defining their expected behavior, even if the likelihood is low.

The documentation quality is high. The Ajna Finance team provided a whitepaper describing the full protocol, including the grant system, as well as a master specification document with more technical details. Voting code follows good code patterns and it is implemented as described in the specification. However, the auditing team found some discrepancies between these documents and the codebase (Extraordinary Funding). We ask the Ajna Finance team to fix these inconsistencies in AJN-12 adding more detail to the related sections of the documentation.

ID	DESCRIPTION	SEVERITY	STATUS
AJN-1	Delegate Rewards Calculation Takes Into Account All Funding Voters	● High	Fixed
AJN-2	Locked Funds Due to Non-Executable Proposals	● High	Fixed
AJN-3	Impact of Unaccounted Delegation Rewards on Treasury and GBC Calculation	● High	Fixed
AJN-4	Gas Usage / Loop Concerns	● Low	Acknowledged
AJN-5	Missing Input Validation	● Low	Fixed
AJN-6	Inconsistent Time Period	● Low	Fixed
AJN-7	Zero ERC20 Transfer	● Low	Fixed
AJN-8	Old Version of Solidity	● Informational	Fixed
AJN-9	Multiple Possible 0-1 Knapsack Solutions, but only the First One Gets Accepted	● Informational	Fixed
AJN-10	Outdated Proposal Hashing Function	● Informational	Fixed
AJN-11	Greedy Contract	● Informational	Acknowledged
AJN-12	Extraordinary Funding Proposal Success Condition Discrepancies	● Undetermined	Fixed
AJN-13	Success of the Extraordinary Funding Is Susceptible to Treasury Balance Variability	● Undetermined	Fixed

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Disclaimer
Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following

    1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
    2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
    3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
    1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
    2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Findings

AJN-1 Delegate Rewards Calculation Takes Into Account All Funding Voters

● High ⓘ Fixed

Update
Marked as "Fixed" by the client. Addressed in: `28de025759692d479252df537c2a861fa495ec0e`.

File(s) affected: `StandardFunction.sol`

Description: Delegate rewards are given when a user with delegated votes has voted in both screening and funding rounds. `_getDelegateReward()` function calculates the number of rewards the user should receive. It uses the following function:

$$\text{rewards} = \frac{TotalFundsAvailable * UserUsedVotingPower}{TotalCastedFundingVotes} * 0.1$$

This logic works correctly if it is assumed that everyone has also voted in the screening round, which is not necessarily the case. If the user has not voted in the screening round, the function would revert. Hence the rewards are distributed amongst all users that have participated and have not participated in the screening round in the current logic, and some reward amounts will not be accessible to anyone.

Exploit Scenario:
1. Ten users have voted in the funding round, but only two of them have voted in both funding and screening periods.
2. The rewards will be calculated based on the sum of funding votes of all of the ten users, not just the eligible two users.

Recommendation: The protocol should account for users that have not voted in the screening round and hence should not be entitled to the rewards.

AJN-2 Locked Funds Due to Non-Executable Proposals

● High ⓘ Fixed

Update
Marked as "Fixed" by the client. The Ajna team fixed the issue by validating that the destination address is not zero or the AJNA token contract. Addressed in: `79d58000231949d63c0a9e98acf7c24b611d0d2c`.

File(s) affected: `StandardFunding.sol`

Description: AJNA tokens may become locked in this contract in the case of a Standard Proposal failing to execute. `Funding._execute()` will revert if transferring AJNA tokens to any of the target addresses included in a proposal fails. In `StandardFunding._updateTreasury()` the treasury is updated by only considering the tokens requested for a proposal, regardless of the status of the proposal execution. As a result, upon the failed execution of a Standard Proposal, the treasury will not be updated to include these funds which could not be sent.

Recommendation: Instead of simply reverting when there is a failed execution of a proposal, refund the funds allocated to that proposal to the Ajna treasury.

AJN-3 Impact of Unaccounted Delegation Rewards on Treasury and GBC Calculation

● High ⓘ Fixed

Update
Marked as "Fixed" by the client. Addressed in: `44785d86235f3cd5ceadf77a5128145bfd56ed61`.

File(s) affected: `StandardFunding.sol`

Description: A new standard funding distribution period starts through the execution of `startNewDistributionPeriod()` , , resulting in the automatic deduction of the global budget constraint (GBC) from the available balance in the `treasury` .

When a distribution period expires and `startNewDistributionPeriod()` is called to start a new distribution period, `_updateTreasury()` will add back to `treasury` all the unused funds of the previous periods.

However, the contract does not maintain a record of the funds allocated for delegation rewards (10% of the GBC intended for rewarding eligible voters via the `claimDelegateReward()` function). `_updateTreasury()` will add this 10% back to the treasury, independently of if the rewards were claimed or not. As a result, the `treasury` variable surpasses the contract balance, leading to an inaccurate calculation of the GBC.

Recommendation: Devise a mechanism, perhaps in `_updateTreasury()` , by which the funds allocated towards voters can be recovered if no voters are eligible to claim rewards for a distribution period. Alternatively, to vote in the funding stage, users could be required to have also voted in the screening stage, to remove the possibility of this peculiar behavior and ensure all allocated voting rewards are paid out.

AJN-4 Gas Usage / Loop Concerns

• Low ⓘ Acknowledged

Update

Marked as "Acknowledged" by the client. The client provided the following explanation:

> We don't want to be too restrictive on how people use the grant fund, and also any costs or proposal failures would be borne by the those creating / executing the proposal and not the rest of the system. If the tokens become unavailable, they have been essentially burned

File(s) affected: `StandardFunding.sol` , `ExtraordinaryFunding.sol`

Related Issue(s): SWC-126, SWC-134

Description: Gas usage is a significant concern for smart contract developers and users, since high gas costs may deter users using the smart contract. Even worse, some gas usage issues may prevent the contract from providing services entirely. For example, if a loop requires too much gas to finish processing, then it may entirely prevent the contract from functioning correctly.

When new proposals are submitted via either `proposeExtraordinary()` or `proposeStandard()` arrays of actions are passed in as parameters for that proposal. It is possible that the arrays will be too large and the transaction will run out of gas. This will only be known when the proposals pass/win the voting and contract attempts to execute them.

Recommendation: Consider adding a limit of array sizes for the proposals. Set and enforce a reasonable limit in `_validateCalldatas()` on the number of allowed transfers that are part of a proposal.

AJN-5 Missing Input Validation

• Low ⓘ Fixed

Update

Marked as "Fixed" by the client. Addressed in: `554ebed847ed8d25a7dae0696d2c339458c5aa0a` .

File(s) affected: `StandardFunding.sol` , `ExtraordinaryFunding.sol`

Related Issue(s): SWC-123

Description: It is important to validate inputs, even if they only come from trusted addresses, to avoid human error. Missing input validation was found in the following scenarios:

1. During the funding stage, users that have AJNA tokens can vote in a quadratic manner. It is possible to cast a vote of 0 power. This can be done accidentally or intentionally. Accidental input of 0 has a worse outcome as due to the current design of `_fundingVote()` , a secondary vote cannot change the direction of the previous vote, and 0 counts as a `support=1` vote. So, if the user wanted to vote again against the proposal, it will not be possible anymore.
2. In `proposeStandard()` `description_` can be an empty string.
3. `proposeExtraordinary()` allows a user to propose an extraordinary vote for a certain proposal. This is only possible to do nine times as the minimum threshold for votes goes up by 5% every time. It is possible to make a proposal with zero tokens requested which will force an extra one passed proposal (the proposal has to be voted on and eligible to be executed). After that `proposeExtraordinary()` will revert on any call as `Maths.WAD - _getMinimumThresholdPercentage()` will overflow.

Recommendation:

1. Check that `voteParams_.votesUsed != 0` .
2. Check that `description_` is not an empty string.
3. Consider adding a check that makes sure that the proposal request amount is bigger than 0.

AJN-6 Inconsistent Time Period

● Low ⓘ Fixed

Update
Marked as "Fixed" by the client. Addressed in: `6415e30ebdfe9976aecad1bedd6f68a041ad43b2`.

File(s) affected: `StandardFunding.sol`

Description: `StandardFunding.claimDelegateReward()` checks if the challenge stage of the distribution has ended before transferring the delegation rewards in this statement:

```
// Check if Challenge Period is still active
if(block.number < _getChallengeStageEndBlock(currentDistribution.endBlock)) revert
ChallengePeriodNotEnded();
```

For that logic, the end block of the challenge stage will be valid, which is not consistent with the validations done in other parts of the code. To follow the patterns in the codebase, users should not be able to claim rewards in the end block of the challenge stage, as it is still in progress.

Recommendation: Avoid claiming delegation rewards when the challenge state has not finished (change the operator to `<=`).

AJN-7 Zero ERC20 Transfer

● Low ⓘ Fixed

Update
Marked as "Fixed" by the client. Addressed in: `86ddbf80499935357954b01d094a3520e3610b8c`.

File(s) affected: `StandardFunding.sol`

Description: `claimDelegateReward()` does not check that `rewardClaimed_` is non-zero prior to transferring to the recipient. In the case where the caller doesn't have any rewards to claim, gas will be wasted on transferring a zero value.

Recommendation: We recommend only invoking the transfer if `rewardClaimed_` is non-zero.

AJN-8 Old Version of Solidity

● Informational ⓘ Fixed

Update
Marked as "Fixed" by the client. Addressed in: `88c81b80c2d0380a3ca7a137988eb5ab75ee49b9`.

Description: The contracts are currently set to be compiled with Solidity version `0.8.16` according to their `pragma` statements. However, an important bug was fixed in `0.8.17`.

Recommendation: Consider upgrading to the recommended solidity version of `0.8.18`.

AJN-9
Multiple Possible 0-1 Knapsack Solutions, but only the First One Gets Accepted

● Informational ⓘ Fixed

Update
Marked as "Acknowledged" by the client. The Ajna team updated the whitepaper to inform the users about this process. The client provided the following explanation:

> Updated documentation

File(s) affected: `StandardFunding.sol`

Description: After the funding part of the voting in Standard Proposal Voting is over, one can submit a winner slate. A winner slate is a selection of winning proposals in such a way that the total funds needed are lower than 90% of GBC. These slates can be submitted only if they are better than the previous one. The ranking is done in a way of the total number of votes in the slate.

This leads to a classic optimization problem called 0-1 Knapsack, which can be defined as follows:

$$\text{maximize} \quad \sum_{i=1}^{10} v_i x_i$$

subject to $\sum_{i=1}^{10} w_i x_i \leq GBC$ and $x_i \in 0, 1$ and $w_i > 0$

As the field is not huge, it is possible to brute force an optimized solution. The problem arises if an optimum solution (i.e. slate with a max number of votes) has multiple combinations. This means there is now another level of selection where the first person to submit an optimal slate is chosen.

Recommendation: Consider also taking into account the length of the slate, not just the number of votes.

AJN-10 Outdated Proposal Hashing Function

● Informational Fixed

Update
Marked as "Fixed" by the client. Addressed in: `988e5ef5367fc36effc0a555a5a44df5e05908ec` .

File(s) affected: `GrantFund.sol` , `IGrantFund.sol`

Description: `GrantFund.hashProposal()` allow users to calculate the proposal ID passing the proposal data as parameters. However, it does not include the prefixes added to the description, and the NatSpec comment about `descriptionHash_` can be confusing as it is not updated for this code commit.

Recommendation: `GrantFund.hashProposal()` should explain how to generate the proposal IDs for each funding type (standard or extraordinary). Depending on the funding method, `description` must contain a different prefix. Another solution is creating two different functions to hash both proposal formats (standard or extraordinary).

AJN-11 Greedy Contract

● Informational Acknowledged

Update
Marked as "Acknowledged" by the client. The client provided the following explanation:

> We acknowledge but don't plan on updating documentation.

Description: A greedy contract is a contract that can receive assets that can never be redeemed. In this case (as in many other smart contracts) if a user transfers ERC20 tokens directly to any of the Ajna Governance contract addresses, the tokens will not be accounted for and will be blocked in the smart contract.

Recommendation: This is a common issue across the majority of the protocols. The design is valid, but the documentation should include some warning for the final user to avoid this situation. User should only use `fundTreasury()` instead of transferring directly.

AJN-12 Extraordinary Funding Proposal Success Condition Discrepancies

● Undetermined Fixed

Update
Marked as "Fixed" by the client. Addressed in: `07b4ae3b6c35f6bff428b3cef559a788be75885e` . The client provided the following explanation:

> Removed Extraordinary Funding

File(s) affected: `ExtraordinaryFunding.sol`

Description: In the Ajna Finance Whitepaper, one of the conditions for an extraordinary funding proposal's success is that the threshold number of required votes is the sum of the "minimum threshold" and the "proposal threshold", or in the terms of the whitepaper:

> *When submitting a proposal the proposer must include the exact number of tokens they would like to extract, proposal threshold. This proposal threshold is added to the minimum threshold to provide a required votes received threshold. If the vote fails to reach this threshold it will fail and no tokens will be distributed.*

The example given treats both the minimum threshold and the proposal threshold as determined in terms of the non-treasury tokens.

a. *A proposer requests tokens equivalent to 10% of the treasury.*

   i. *50% + 10% = 60%*

   ii. *If 65% of non-treasury tokens vote affirmatively, 10% of the treasury is released*

iii. *If 59.9% of non-treasury tokens vote affirmatively, 0% of the treasury is released*

However, as implemented, the proposal threshold is being replaced with the number of tokens requested, as can be seen at `ExtraordinaryFunding.sol#173`:

```
(votesReceived >= tokensRequested_ + _getSliceOfNonTreasury(minThresholdPercentage))
```

This does not agree with the example from the whitepaper. Take the following scenario for example:

1. The treasury contains 300 million tokens.
2. There are 200 million non-treasury tokens.
3. The extraordinary funding proposal requests 10% of the treasury (30 million).
4. The number of votes required based on the description is 120 million votes.
5. The number of votes required based on the implementation is 130 million votes.

There is another ambiguity regarding extraordinary funding proposal success. The following condition is described in the whitepaper:

> *Also, a proposal must not exceed the minimum threshold in what it withdraws. For example, if a proposal specifies that it would like to take 100,000,000 tokens from the treasury, which represents 60% of the treasury supply, and the minimum threshold is 55%, this proposal will automatically fail.*

However, the example that follows is inconsistent with this description:

> *EF2 should now fail since 120,000,000 > ((1 - minimum threshold) * treasury)*

That is, the example demonstrates that a proposal must not exceed (1 - minimum threshold) in the withdrawal amount, whereas the preceding paragraph states that the withdrawal amount must not exceed the minimum threshold. The code implements the example's calculation:

```
(tokensRequested_ <= _getSliceOfTreasury(Maths.WAD - minThresholdPercentage))
```

Finally, point 3 defines the minimum threshold as a proportion of non-treasury tokens, whereas the example in point 5 regards the minimum threshold as a proportion of treasury tokens.

> *This mechanism works by allowing up to the percentage of non-treasury tokens, minimum threshold ... EF2 should now fail since 120,000,000 > ((1 - minimum threshold) * treasury)*

Recommendation: Consider the discrepancies in the descriptions and implementations above and adjust the code and/or whitepaper accordingly.

AJN-13

Success of the Extraordinary Funding Is Susceptible to Treasury Balance Variability

● Undetermined ⓘ Fixed

Update

Marked as "Fixed" by the client. Addressed in: `07b4ae3b6c35f6bff428b3cef559a788be75885e`. The client provided the following explanation:

> Removed Extraordinary Funding

File(s) affected: `ExtraordinaryFunding.sol`

Description: The `executeExtraordinary()` function evaluates if the proposal has passed and is ready to be executed. `_extraordinaryProposalSucceeded()` is the function that actually whether a proposal succeeded or not. The calculation is based on the current state of the treasury and it is verified that at least a certain threshold of the current holdings of the treasury are met.

This means that from the moment of proposal to execution, the state of the treasury can vary significantly. Though no direct vulnerability has been identified, this behavior can lead to unexpected outcomes.

Recommendation: Consider the impact of this in Extraordinary Funding and include necessary treasury balance checks if necessary.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.

- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
- **Informational** – The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Code Documentation

1. Fixed Some typographical errors were found in the codebase. We recommend using a spell checker:
   - `StandardFunding.sol`: `readd -> re-add`.
2. Fixed `ExtraordinaryFunding.sol` shows unclear comments:
   - `// succeeded if proposal's votes received doesn't exceed the minimum threshold required`
   - `// succeeded if tokens requested are available for claiming from the treasury`

Adherence to Best Practices

1. Acknowledged `StandardFunding._fundingVote` should better name the parameter `voteParams_` to `voteParam` as it is a singular value.
2. Fixed `StandardFunding._insertionSortProposalsByVotes` should better name the parameter `_targetProposalId` to better represent its purpose in the function as an index.
3. Fixed In the `Maths` library, there is a `abs()` function that returns the absolute size of a signed integer. It will always be positive, so it makes sense to return it as `uint256` instead of `int`.
4. Acknowledged `IERC20Metadata` is imported as a library in the `BurnWrappedAjna` contracts but not used as a dependency in the inheritance. Confirm if this library is needed. If not, remove the import from `BurnWrapper.sol`.
5. Acknowledged `_topTenProposals, _quadraticVoters,_isSurplusFundsUpdated, hasClaimedReward, screeningVotesCast` take in `uint256` as the index, but `distribution.id` is a `uint24` in `StandardFunding.sol`
6. Fixed `Maths.sol` library should use `WAD` instead of hardcoding the powers.
7. Fixed `proposal.startBlock > block.number` check is redundant in `L139` of `ExtraordinaryFunding.sol`
8. Fixed Declare `uint256` instead of `uint`. **Update:** Some for loops are still declaring `uint`.
9. Fixed To avoid code duplication, `ExtraordinaryFunding.executeExtraordinary()` could call `_getExtraordinaryProposalState()` in `L70`.
10. Fixed To avoid code duplication, `StandardFunding.executeStandard()` could call `_standardProposalState()` in `L358`.
11. Fixed `_standardProposalState()` can be renamed to `_getExtraordinaryProposalState()` or viceversa to be consistent in both contracts.
12. Fixed The error `InsufficientVotingPower` in `L659` of `StandardFunding.sol`, is not descriptive for that case. Consider creating another error with a different name.
13. Some minor improvements to `StandardFunding.sol`:
    1. Fixed `L282` Remove this check as it is unnecessary -- would return zero anyways in the calculation. Including this check merely adds gas to the average claimer's transaction.
    2. Fixed `L317` The check can be simplified to `sum > _sumProposalFundingVotes(_fundedProposalSlates[currentSlateHash])`.
    3. Fixed `L448` The check can be moved after the `for` loop.

Adherence to Specification

1. Fixed The white paper does not mention that the voting power for the screening and funding stage is based on the minimum value of the voter's votes at 33 blocks prior to the period and the votes held at the start of the period. The whitepaper only mentions that the balance will be from 33 blocks prior to the start block. The documentation and the code should align.
2. Acknowledged The documentation mentions that 2% of the treasury will be distributed to projects, however, the code allows 3% to be allocated for distribution. These values should align. **Update:** The Ajna team will update the whitepaper to a new version.
3. Unresolved `GrantFund.sol#L258` The comment should be updated because we no longer wait one week after the distribution period ends.
4. Unresolved `GrantFund.sol#L289` The comment should be changed as the logic for calculating the screening stage end has changed.

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Contracts

- `bde...009 ./src/grants/GrantFund.sol`
- `a81...d5c ./src/grants/interfaces/IStandardFunding.sol`
- `30a...b58 ./src/grants/interfaces/IExtraordinaryFunding.sol`
- `1e5...1fa ./src/grants/interfaces/IGrantFund.sol`
- `231...428 ./src/grants/interfaces/IFunding.sol`
- `303...3fe ./src/grants/base/StandardFunding.sol`
- `ba7...ec9 ./src/grants/base/ExtraordinaryFunding.sol`
- `8ac...01b ./src/grants/base/Funding.sol`
- `182...b2a ./src/grants/libraries/Maths.sol`
- `42c...098 ./src/token/AjnaToken.sol`
- `765...156 ./src/token/BurnWrapper.sol`

Tests

- `fb6...c8b ./test/utils/GrantFundTestHelper.sol`
- `e42...c1c ./test/utils/IAjnaToken.sol`
- `e1d...1ed ./test/utils/SigUtils.sol`
- `d4b...dfa ./test/invariants/StandardFundingInvariant.t.sol`
- `fe3...427 ./test/invariants/ExtraordinaryInvariant.t.sol`
- `5db...ff6 ./test/invariants/StandardMultipleDistributionInvariant.t.sol`
- `e29...7fe ./test/invariants/StandardScreeningInvariant.t.sol`
- `a9b...27a ./test/invariants/StandardFinalizeInvariant.t.sol`
- `1e8...375 ./test/invariants/base/ExtraordinaryTestBase.sol`
- `e8f...8de ./test/invariants/base/StandardTestBase.sol`
- `093...87d ./test/invariants/base/TestBase.sol`
- `bc3...13a ./test/invariants/base/ITestBase.sol`
- `09e...e0a ./test/invariants/handlers/Handler.sol`
- `8c7...5cb ./test/invariants/handlers/ExtraordinaryHandler.sol`
- `cc4...ce2 ./test/invariants/handlers/StandardHandler.sol`
- `1e0...37c ./test/unit/GrantFund.t.sol`
- `39c...e51 ./test/unit/ExtraordinaryFunding.t.sol`
- `cba...7d4 ./test/unit/AjnaToken.t.sol`
- `e1b...4d0 ./test/unit/StandardFunding.t.sol`
- `527...648 ./test/unit/BurnWrappedToken.t.sol`
- `887...6eb ./test/unit/Maths.t.sol`
- `654...d37 ./test/interactions/DrainGrantFund.sol`

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:
- Slither v0.9.3

Steps taken to run the tools:
1. Install the Slither tool: `pip3 install slither-analyzer`
2. Run Slither from the project directory: `slither .`

Automated Analysis

Slither

Slither found 884 potential issues. Most of them were found in the test files and marked as false positives. All of them were discussed in this report or classified as false positives.

Test Suite Results

All tests passed.
Update: New tests were added to cover fixes and the removal of the Extraordinary Funding Method. All tests passed.

```
forge clean && forge test --mt test --optimize --optimizer-runs 1000000 -v # --ffi # enable if you need
the `ffi` cheat code on HEVM
[⠆] Compiling...
[⠰] Compiling 60 files with 0.8.18
[⠆] Compiling 42 files with 0.8.7
[⠃] Solc 0.8.7 finished in 2.85s
[⠰] Solc 0.8.18 finished in 8.59s
Compiler run successful (with warnings)
warning[9432]: Warning: "prevrandao" is not supported by the VM version and will be treated as
"difficulty".
   --> test/utils/GrantFundTestHelper.sol:333:70:
    |
333 |         proposal_ = uint256(keccak256(abi.encodePacked(block.number, block.prevrandao))) %
noOfProposals_;
    |                                                                      ^^^^^^^^^^^^^^^^



warning[9432]: Warning: "prevrandao" is not supported by the VM version and will be treated as
"difficulty".
   --> test/utils/GrantFundTestHelper.sol:412:84:
    |
412 |             uint256 randomNonce = uint256(keccak256(abi.encodePacked(block.number,
block.prevrandao))) % 100;
    |                                                                                    ^^^^^^^^^^^^^^^^



warning[9432]: Warning: "prevrandao" is not supported by the VM version and will be treated as
"difficulty".
   --> test/utils/GrantFundTestHelper.sol:606:71:
    |
606 |         votes_ = 1 + uint256(keccak256(abi.encodePacked(block.number, block.prevrandao))) % (1.25 *
1e18);
    |                                                                       ^^^^^^^^^^^^^^^^



warning[9432]: Warning: "prevrandao" is not supported by the VM version and will be treated as
"difficulty".
   --> test/invariants/handlers/Handler.sol:210:65:
    |
210 |         return uint256(keccak256(abi.encodePacked(block.number, block.prevrandao, counter)));
    |                                                                 ^^^^^^^^^^^^^^^^




Running 5 tests for test/unit/Maths.t.sol:MathsTest
[PASS] testAbs() (gas: 1513)
[PASS] testDivision() (gas: 1352)
[PASS] testMin() (gas: 577)
[PASS] testMultiplication() (gas: 1790)
[PASS] testPow() (gas: 8150)
Test result: ok. 5 passed; 0 failed; finished in 4.07ms

Running 3 tests for test/unit/GrantFund.t.sol:GrantFundTest
[PASS] testFundTreasury() (gas: 81776)
[PASS] testTreasuryDistributionPeriodFunding() (gas: 138384)
```

```
[PASS] testTreasuryInsufficientBalanceStandard() (gas: 298228)
Test result: ok. 3 passed; 0 failed; finished in 5.35ms

Running 5 tests for test/unit/BurnWrappedToken.t.sol:BurnWrappedTokenTest
[PASS] testBaseInvariantMetadata() (gas: 15974)
[PASS] testCantUnwrap() (gas: 148653)
[PASS] testOnlyWrapAjna() (gas: 732962)
[PASS] testWrap() (gas: 170649)
[PASS] testWrappedInvariantMetadata() (gas: 15877)
Test result: ok. 5 passed; 0 failed; finished in 2.16s

Running 11 tests for test/unit/AjnaToken.t.sol:AjnaTokenTest
[PASS] testBaseInvariantMetadata() (gas: 15956)
[PASS] testBurn() (gas: 34364)
[PASS] testCalculateVotingPower() (gas: 738529)
[PASS] testCannotSendTokensToContract() (gas: 8872)
[PASS] testDelegateVotes() (gas: 306673)
[PASS] testHolderTokenBalance() (gas: 12126)
[PASS] testNestedDelegation() (gas: 185430)
[PASS] testTokenTotalSupply() (gas: 9825)
[PASS] testTransferTokensToZeroAddress() (gas: 8845)
[PASS] testTransferWithApprove(uint256) (runs: 150, μ: 60251, ~: 60251)
[PASS] testTransferWithPermit(uint256) (runs: 150, μ: 377836, ~: 377838)
Test result: ok. 11 passed; 0 failed; finished in 19.06s

Running 18 tests for test/unit/StandardFunding.t.sol:StandardFundingGrantFundTest
[PASS] testDistributionPeriodEndToEnd() (gas: 5234473)
[PASS] testFuzzTopTenProposalandDelegateReward(uint256,uint256) (runs: 150, μ: 83130489, ~: 76247234)
[PASS] testGetVotingPowerFundingStage() (gas: 1753184)
[PASS] testGetVotingPowerScreeningStage() (gas: 1692906)
[PASS] testInvalidProposalCalldata() (gas: 1509999)
[PASS] testInvalidProposalCalldataSelector() (gas: 104229)
[PASS] testInvalidProposalDescription() (gas: 58955)
[PASS] testInvalidProposalTarget() (gas: 155239)
[PASS] testMultipleDistribution() (gas: 3063618)
[PASS] testPropose() (gas: 301681)
[PASS] testQuadraticVotingTally() (gas: 1939378)
[PASS] testScreenProposalsCheckSorting() (gas: 4140047)
[PASS] testScreenProposalsMulti() (gas: 3695800)
[PASS] testSingleRoundVoting() (gas: 3164221)
[PASS] testStage() (gas: 1331759)
[PASS] testStartNewDistributionPeriod() (gas: 119415)
[PASS] testTreasuryUpdatedWithSurplusTokens() (gas: 2498163)
[PASS] testVotesCast() (gas: 1900503)
Test result: ok. 18 passed; 0 failed; finished in 19.06s
```

Code Coverage

The project implements code coverage and shows good metrics.

Update: New tests were added that improved code coverage.

Note from Ajna Finance: *There are two branches in GrantFund.sol that show as unreached in our tests. Manual search shows this to not be the case, with the branches checked at StandardFunding.t.sol#821, and StandardFunding.t.sol#1151. The missing branches can be seen with our ./check-code-coverage.sh script*

File	% Lines	% Functions	% Branches
src/grants/GrantFund.sol	100.0% (**280**/280)	100.0% (**47**/47)	98.1% (**102**/104)
src/grants/libraries/Maths.sol	100.0% (**9**/9)	100.0% (**5**/5)	100.0% (**2**/2)
src/token/AjnaToken.sol	83.3% (**5**/6)	80.0% (**4**/5)	100.0% (**2**/2)
src/token/BurnWrapper.sol	100.0% (**2**/2)	100.00% (**2**/2)	100.0% (**0**/0)

File	% Lines	% Functions	% Branches
Total	99.7% (**296/**297)	98.3% (**58/**59)	98.1% (**106/**108)

Changelog

- 2023-05-04 - Initial report
- 2023-05-23 - Updated the report according to commit `07b4ae3`
- 2023-05-31 - Updated the report according to commit `24cf70c`

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT