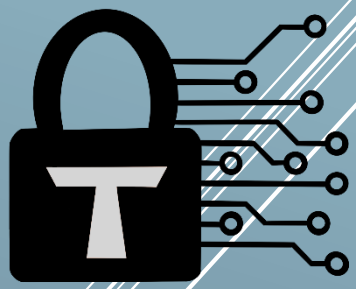
Trust Security



Smart Contract Audit

Agora Governance

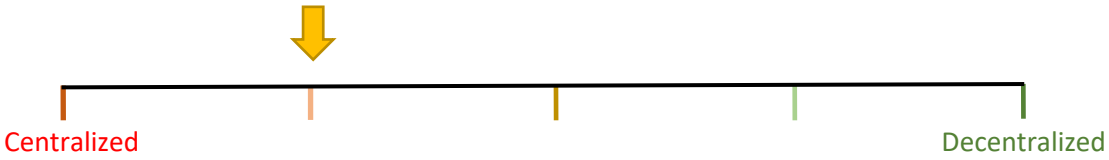
02/08/24

Executive summary

Category	Governance
Audited file count	8
Lines of Code	955
Auditor	rvierdiiev, 100proof, ether_sky
Time period	17/06-29/06/24

Severity	Total	Fixed
High	1	1
Medium	3	3
Low	1	1

Centralization score



SIGNATURE

Digitally signed by Trust
Date: 2024.08.02 12:02:33+03'00'

Document properties

Versioning

Version	Date	Description
0.1	29/06/24	Client report
0.2	23/07/24	Mitigation review
0.3	02/07/24	Final review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- src/AgoraGovernor.sol
- src/ProxyAdmin.sol
- src/TokenDistributor.sol
- src/AgoraTimelock.sol
- src/ProposalTypesConfigurator.sol
- src/modules/VotingModule.sol
- src/modules/ApprovalVotingModule.sol
- src/modules/OptimisticModule.sol

Repository details

- **Repository URL:** https://github.com/voteagora/agora-governor
- **Commit hash:** 286b6425bfcb76553170f2cac320f42f4ad67eda
- **Mitigation review hash:** e702019f259f236062855f37abf12b95b310746e
- **Final review hash:** 2c3d4a6c11a45d43d1e489fc00406e38df67ee0a

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

100proof transitioned into smart contract security after many years as a software developer. Starting out on Code4rena he has developed experience in a number of protocol categories: AMMs/CLAMMs, Options/Perpetuals, Lending, and Governance. He is also a prolific bug bounty hunter, scoring 7-figures in public bounty hunting.

rvierdiiev is a Web3 security researcher who participated in many public audit contests on multiple platforms and has a proven track record of experience. He achieved the #1 spot on leaderboards many times over the years.

ether_sky is a security researcher with a focus on blockchain security. He specializes in algorithms and data structures and has a solid background in IT development. He has placed at the top of audit contests in Code4rena and Sherlock recently.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	The code is well modularized to reduce complexity.
Documentation	**Moderate**	While some inline documentation is provided, more comprehensive docs would be more ideal.
Best practices	**Good**	Project consistently adheres to industry standards.
Centralization risks	**Moderate**	Project introduces several points of centralization vectors (upgradeability, admin multisig and others)

Findings

High severity findings

TRST-H-1 Proposals for ApprovalVotingModule can pass with zero quorum and for-votes in presence of OptimisticModule

- **Category:** Logical flaws
- **Source:** ApprovalVotingModule.sol
- **Status:** Fixed

Description

Agora has introduced the concept of proposal types which allow multiple configurations of *quorum* and *approval threshold*. Proposals for the OptimisticModule require that both quorum and approval threshold are zero. To be able to propose for the **OptimisticModule** requires such a proposal type exists.

However, if it exists, then there is nothing stopping a user from making an **ApprovalVotingModule** proposal using this proposal type.

The impact is that such proposals require no quorum and always succeed regardless of voting, even though their impact may be very high (e.g. stealing funds, changing important governance parameters). There is no limit to how many times this can be done and the only current solutions are:

- an administrator cancels the proposal with *AgoraGovernor.cancel()*
- an administrator edits the proposal type with *AgoraGovernor.editProposalType()*
- *ProposalTypeConfigurator.setProposalType()* is called to change the *quorum/approvalThreshold.*

The first two solutions are insufficient as they require active monitoring of all proposals and the third one has the side effect of disabling the **OptimisticModule** from being used.

Recommended mitigation

Two possible mitigations are as follows.

1. Add checks for reasonable quorum and approval threshold values in *ApprovalVotingModule.propose()* after L102:

```
(ProposalOption[] memory proposalOptions, ProposalSettings memory proposalSettings) =
    abi.decode(proposalData, (ProposalOption[], ProposalSettings));

uint8 proposalTypeId = IAgoraGovernor(msg.sender).getProposalType(proposalId);
IProposalTypesConfigurator proposalConfigurator =
  IProposalTypesConfigurator(IAgoraGovernor(msg.sender).PROPOSAL_TYPES_CONFIGURATOR());
IProposalTypesConfigurator.ProposalType memory proposalType =
  proposalConfigurator.proposalTypes(proposalTypeId);

if (proposalType.approvalThreshold <= 5000) revert NotApprovalVotingProposalType();
```

2. Add a new *address votingModule* field to the ProposalType struct and check for valid values inside each voting module's *propose()* function, as below:

```
if (proposalType.votingModule != address(this)) revert InvalidVotingModule();
```

Team response

Fixed

Mitigation Review

The fix is insufficient as *proposeWithModule()* allows a proposal type with *module == address(0)* to be provided without a revert occurring. The relevant code snippet is:

```
if (
    bytes(PROPOSAL_TYPES_CONFIGURATOR.proposalTypes(proposalType).name).length == 0
        || (
            PROPOSAL_TYPES_CONFIGURATOR.proposalTypes(proposalType).module != address(0)
                && PROPOSAL_TYPES_CONFIGURATOR.proposalTypes(proposalType).module !=
address(module)
        )
) {
    revert InvalidProposalType(proposalType);
}
```

Proposal types for voting modules should not allow *module == address(0).*

Team response

Fixed.

Mitigation Review

The fix corrects the issue.

Medium severity findings

TRST-M-1 AgoraGovernor balance is checked instead of AgoraTimelock, which allows proposal to steal governance funds

- **Category:** Logical flaws
- **Source:** ApprovalVotingModule.sol
- **Status:** Fixed

Description

ApprovingVotingModule allows users to vote on different options. One or more of them are selected as winners at the end of voting to be executed on behalf of **AgoraTimelock**. Once executed, the proposal can use some funds from Timelock. It can be a native token or ERC20 token. In case of an ERC20 token, the address of the token that will be used is provided in the **budgetToken** field.

The protocol tries to control the amount of funds that will be spent by the proposal and not let the proposal use more than the **budgetAmount** parameter. To do so the balance of the AgoraGovernance contract is stored to the proposal info in the *ApprovingVotingModule._formatExecuteParams()* function before execution of the proposal and then later the *ApprovingVotingModule._afterExcute()* function, which is called as the last target of the proposal, checks that no more than **budgetAmount** was spent.

But because all funds are planned to be collected in the **AgoraTimelock** contract instead of **AgoraGovernance**, this check won't work and will not be able to detect theft of ERC20 tokens.

Recommended mitigation

Fetch the balance of **AgoraTimelock** instead of **AgoraGovernance**.

Team response

Fixed.

Mitigation Review

The fix corrects the issue.

TRST-M-2 In the ApprovalVotingModule, options with 0 votes can be executed

- **Category:** Logical flaws
- **Source:** ApprovalVotingModule.sol
- **Status:** Fixed

Description

Imagine a proposer creates a proposal with the criteria set to **PassingCriteria.TopChoices** and a criteria value of 2. During the voting period, the voters only vote for the first option. Since one option has non-zero votes, this module is considered successful.

```
function _voteSucceeded(uint256 proposalId) external view override returns (bool) {
    Proposal memory proposal = proposals[proposalId];
    ProposalOption[] memory options = proposal.options;
    uint256 n = options.length;
    unchecked {
        if (proposal.settings.criteria == uint8(PassingCriteria.Threshold)) {
            for (uint256 i; i < n; ++i) {
                if (proposal.optionVotes[i] >= proposal.settings.criteriaValue) return true;
            }
        } else if (proposal.settings.criteria == uint8(PassingCriteria.TopChoices)) {
            for (uint256 i; i < n; ++i) {
                if (proposal.optionVotes[i] != 0) return true;
            }
        }
    }

    return false;
}
```

However, in the **_countOptions** function, we include the first two options, even though the second option has 0 votes.

```
function _countOptions(
    ProposalOption[] memory options,
    uint128[] memory optionVotes,
    ProposalSettings memory settings
) internal pure returns (uint256 executeParamsLength, uint256 succeededOptionsLength) {
    uint256 n = options.length;
    unchecked {
        uint256 i;
        if (settings.criteria == uint8(PassingCriteria.Threshold)) {
            for (i; i < n; ++i) {
                if (optionVotes[i] >= settings.criteriaValue) {
                    executeParamsLength += options[i].targets.length;
                } else {
                    break;
                }
            }
        } else if (settings.criteria == uint8(PassingCriteria.TopChoices)) {
            for (i; i < settings.criteriaValue; ++i) {
                executeParamsLength += options[i].targets.length;
            }
        }
        succeededOptionsLength = i;
    }
}
```

Obviously, the voters didn't vote for the second option because they didn't want it to be executed.

However, this second option will also be executed.

Recommended mitigation

Only include options with non-zero votes in the **_countOptions** function.

```
function _countOptions(
    ProposalOption[] memory options,
    uint128[] memory optionVotes,
    ProposalSettings memory settings
) internal pure returns (uint256 executeParamsLength, uint256 succeededOptionsLength) {
    uint256 n = options.length;
    unchecked {
        uint256 i;
        if (settings.criteria == uint8(PassingCriteria.Threshold)) {
            for (i; i < n; ++i) {
                if (optionVotes[i] >= settings.criteriaValue) {
                    executeParamsLength += options[i].targets.length;
                } else {
                    break;
                }
            }
        } else if (settings.criteria == uint8(PassingCriteria.TopChoices)) {
            for (i; i < settings.criteriaValue; ++i) {
-                executeParamsLength += options[i].targets.length;
+                if (optionVotes[i] > 0) {
+                    executeParamsLength += options[i].targets.length;
+                } else {
+                    break;
+                }
            }
        }
        succeededOptionsLength = i;
    }
}
```

Alternatively, consider the module successful only when the first **criteriaValue** options have non-zero votes.

Team response

Fixed.

Mitigation Review

The fix commit addresses the issue.

TRST-M-3 In the ApprovalVotingModule, the proposal can use more tokens than are allowed

- **Category:** Logical flaws
- **Source:** ApprovalVotingModule.sol
- **Status:** Fixed

Description

In the **ApprovalVotingModule**, each option has a **budgetTokensSpent** value to specify the allowed token amount spent on that option, and the proposal has a **budgetAmount** value to limit the total token amount spent during its execution when **budgetToken** exists. After execution, it verifies the total token expenditure and revert execution if it exceeds **budgetAmount**.

However, it's possible that winning options may exceed their allocated token limits. Imagine a proposal with 3 options and a **budgetAmount** set to 1000 tokens. The **budgetTokensSpent** for these options are 500, 100, and 700 tokens, respectively. All three options are winners, with 100, 90, and 80 votes.

In the **_formatExecuteParams** function, the third option is excluded from execution because the sum of **budgetTokensSpent** reaches 1300 tokens, exceeding the **budgetAmount**. As a result, only the first two options will execute.

```
function _formatExecuteParams(uint256 proposalId, bytes memory proposalData)
    public
    override
    returns (address[] memory targets, uint256[] memory values, bytes[] memory calldatas)
{
    for (uint256 i; i < succeededOptionsLength;) {
        option = sortedOptions[i];
        for (n = 0; n < option.targets.length;) {
            if (settings.budgetAmount != 0) {
                if (settings.budgetToken == address(0)) {
                    if (option.values[n] != 0) totalValue += option.values[n];
                }
                if (totalValue > settings.budgetAmount) break;
            }
            unchecked {
                executeParams[executeParamsLength + n] =
                    ExecuteParams(option.targets[n], option.values[n], option.calldatas[n]);
                ++n;
            }
        }
        if (settings.budgetAmount != 0) {
            if (settings.budgetToken != address(0)) {
                if (option.budgetTokensSpent != 0) totalValue += option.budgetTokensSpent;
            }
            if (totalValue > settings.budgetAmount) break;
        }
        unchecked {
            executeParamsLength += n;
            ++i;
        }
    }
}
```

Note that the second option has a fake **budgetTokenAmount** - voters believe it will spend at most 100 tokens as specified, but it actually spends 500 tokens. The final token expenditure checks pass because the actual spent amount of 1000 tokens doesn't exceed the **budgetAmount** limit.

```
function _afterExecute(uint256 proposalId, bytes memory proposalData) public view {
   (, ProposalSettings memory settings) = abi.decode(proposalData, (ProposalOption[],
ProposalSettings));

    if (settings.budgetToken != address(0)) {
        address governor = proposals[proposalId].governor;
        uint256 initBalance = proposals[proposalId].initBalance;
        uint256 finalBalance = IERC20(settings.budgetToken).balanceOf(governor);
        if (finalBalance < initBalance) {
            unchecked {
                if (initBalance - finalBalance > settings.budgetAmount) revert
BudgetExceeded();
            }
        }
    }
}
```

This scenario illustrates that options can use more tokens than allowed, which could potentially be exploited by malicious users to steal tokens.

Recommended mitigation

Snapshot the last sum of **budgetTokensSpent** before exceeding the **budgetAmount** and insert this as the third parameter into the **_afterExecute** function. This approach helps prevent options from using more tokens than specified by **budgetTokenAmount**.

Team response

Fixed.

Mitigation Review

The fix is insufficient because *totalValue* will be greater than *settings.budgetAmount* if the for-loops in *_formatExecuteParams()* are broken due to the following check:

```
if (totalValue > settings.budgetAmount) break;
```

Team response

Fixed.

Mitigation Review

The fix corrects the issue.

Low severity findings

TRST-L-1 In the ApprovalVotingModule, if budgetAmount is 0 and budgetToken exists, the proposal execution can be reverted

- **Category:** Logical flaws
- **Source:** ApprovalVotingModule.sol
- **Status:** Fixed

Description

In the **ApprovalVotingModule**, a proposal can have a **budgetToken** to indicate token usage during execution, while setting **budgetAmount** to 0 indicates no limit on token expenditure. However, executing such proposals will revert if any tokens are used during execution.

```
function _afterExecute(uint256 proposalId, bytes memory proposalData) public view {
   (, ProposalSettings memory settings) = abi.decode(proposalData, (ProposalOption[],
ProposalSettings));

    if (settings.budgetToken != address(0)) {
        address governor = proposals[proposalId].governor;
        uint256 initBalance = proposals[proposalId].initBalance;
        uint256 finalBalance = IERC20(settings.budgetToken).balanceOf(governor);
        if (finalBalance < initBalance) {
            unchecked {
                if (initBalance - finalBalance > settings.budgetAmount) revert
BudgetExceeded();
            }
        }
    }
}
```

Recommended mitigation

We can perform a final check when **budgetAmount** is not 0 and **budgetToken** exists.

```
function _afterExecute(uint256 proposalId, bytes memory proposalData) public view {
   (, ProposalSettings memory settings) = abi.decode(proposalData, (ProposalOption[],
ProposalSettings));

-    if (settings.budgetToken != address(0)) {
+    if (settings.budgetToken != address(0) && settings.budgetAmount > 0) {
        address governor = proposals[proposalId].governor;
        uint256 initBalance = proposals[proposalId].initBalance;
        uint256 finalBalance = IERC20(settings.budgetToken).balanceOf(governor);
        if (finalBalance < initBalance) {
            unchecked {
                if (initBalance - finalBalance > settings.budgetAmount) revert
BudgetExceeded();
            }
```

```
        }
    }
}
```

Team response

Fixed.

Mitigation Review

The fix corrects the issue.

Additional recommendations

TRST-R-1 Emit event with options that were executed in the proposal

A proposal that is voted on with **ApprovalVotingModule** may contain several options for users to vote. Even when an option was selected by users its execution can be skipped, because of the budget amount check. This makes it unclear for voters, which options were actually executed. We recommend emitting an **ProposalOptionExecuted** event for each option or **ProposalOptionsExecuted** event that will contain information about all executed options.

```
event ProposalOptionExecuted(uint256 budgetTokensSpent, address[] targets, uint256[] values,
bytes[] calldatas, string description, uint256 votes);
```

TRST-R-2 Add documentation to *ProxyAdmin.updateOwner()*

It is possible to remove the last owner using *ProxyAdmin.updateOwner()*. Add some documentation to the function explaining this behavior.

TRST-R-3 Restrict what ApprovalVotingModule can execute

The current implementation of **ApprovalVotingModule** attempts to add restrictions on how much native ETH/tokens can be spent by the budget proposal (as well as adding multiple options that can be independently voted upon).

However, there are many edge cases which have been described in the report above. The general problem is that the amount specified in fields such as *budgetTokensSpent* and *budgetAmount* are not necessarily actually respected in the *calldata* of the options.

To ensure the safety of the module it should be restricted to a limited set of actions and not allow arbitrary calldata.

First, define an *enum* called e.g. *Action* which enumerates all the valid actions.

```
enum Action {
 TransferToken,
 // ... any other actions you want
}
```

Then, for each enum value, define a *struct.* For example:

```
struct ActionTransferToken {
 uint256 tokensSpent;
}
```

Encode the struct in a new field in *option* called *data.*

Finally, within function *_formatExecuteParams(),* generate the correct calldata for that action based on the fields in the *struct:*

```
if (option.action == Action.TokenTransfer) {
 ActionTransferToken memory actionTransferToken =
   abi.decode(option.data, (ActionTransferToken));
  ... // generate calldata for this action based on fields in actionTransferToken
}
```

Centralization risks

TRST-CR-1 AgoraGovernor admin has excessive privileges

The *admin* of AgoraGovernor contract has the ability to call the *setModuleApproval, setProposalDeadline, setVotingDelay, setVotingPeriod, editProposalType, setProposalThreshold, cancel, and cancelWithModule* functions.

Besides *cancel()* and *cancelWithModule()* two more of these functions can be used to modify existing proposals: *setProposalDeadline()* and *editProposalType()*. The *cancel()* functions remove a proposal entirely, ensuring no harmful changes to the protocol can occur, however:

- *setProposalDeadline()* can be used to extend voting indefinitely thus increasing the chance that a proposal will pass
- *editProposalType()* can be used to change the quorum and approval threshold to very low values (even zero).

Thus, harmful proposals such as “rug pulls” are made that much more likely.

We believe that only *cancel()* and *cancelWithModule()* functions should be allowed to be called by admin, and just to be able to cancel malicious proposals. All other functions listed here, should not be available for admin and that functionality should be configured through the proposals by governance.

TRST-CR-2 onlyAdminOrTimelock allows changes that affect existing proposals

OpenZeppelin’s *onlyGovernance* modifier ensures that calls to functions are only executed as part of proposals. This means that functions such as *setVotingPeriod()* cannot be used to affect existing proposals e.g. by setting a very short voting period which then allows proposals made after to finish before existing ones.

However, Agora’s *onlyAdminOrTimelock* modifier allows admin to call many functions immediately, such as *setVotingDelay(), setVotingPeriod(),* and *setProposalDeadline()*, which would affect existing proposals.