AAVE

Aave Governance v3 Contract Review

Version: 4.1

October, 2023

Contents

Introduction **2**
- Disclaimer . . . 2
- Document Structure . . . 2
- Overview . . . 2

Security Assessment Summary **3**
- Findings Summary . . . 3

Detailed Findings **4**

Summary of Findings **5**
- Payloads may not be created with the correct access level . . . 6
- Payloads are not validated during proposal creation . . . 8
- `PayloadController` is unable to receive native tokens . . . 10
- Removing all voting configs may lock the governance contract . . . 11
- The variable `_votingPortalsCount` is not updated correctly . . . 12
- Voting duration is not checked before a proposal is queued or voted on . . . 13
- Small executor grace periods will prevent payload execution . . . 14
- Payload IDs are not re-org safe . . . 15
- Executor reconfiguration allows for execution of expired payloads . . . 16
- Miscellaneous General Comments . . . 17

A Test Suite **18**

B Vulnerability Severity Classification **20**

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the AAVE smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the AAVE smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the AAVE smart contracts.

Overview

Aave Governance V3 is the most recent version of Aave on-chain governance. It allows users to control the entire ecosystem in a decentralised way. The new version is cross-chain and aims to minimise voting cost.

Aave Governance V3 allows users to create proposals on the Governance contract deployed on Ethereum with different payloads that could belong to different networks. When creating the proposal, the proposer specifies the payloads to execute and the networks where these payloads are going to be executed, as well as the *voting network*, where the votes will be cast. Users can then vote on proposals using their voting power on Ethereum via the *voting machine* contract deployed on the voting network.

Security Assessment Summary

This review was conducted on the files hosted on the Aave Governance v3 repository and were assessed at commit *a72fa0e*.

Specifically, the files in scope are as follows:

- `Governance.sol`
- `GovernanceCore.sol`
- `GovernancePowerStrategy.sol`
- `BaseGovernancePowerStrategy.sol`
- `BaseVotingStrategy.sol`
- `VotingPortal.sol`
- `SlotUtils.sol`
- `Executor.sol`
- `PayloadsController.sol`
- `PayloadsControllerCore.sol`
- `DataWarehouse.sol`
- `VotingMachine.sol`
- `VotingMachineWithProofs.sol`
- `VotingStrategy.sol`
- `Errors.sol`

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 10 issues during this assessment. Categorised by their severity:

- High: 1 issue.
- Medium: 2 issues.
- Low: 5 issues.
- Informational: 2 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the AAVE smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
AG3-01	Payloads may not be created with the correct access level	High	Resolved
AG3-02	Payloads are not validated during proposal creation	Medium	Resolved
AG3-03	`PayloadController` is unable to receive native tokens	Medium	Resolved
AG3-04	Removing all voting configs may lock the governance contract	Low	Resolved
AG3-05	The variable `_votingPortalsCount` is not updated correctly	Low	Resolved
AG3-06	Voting duration is not checked before a proposal is queued or voted on	Low	Resolved
AG3-07	Small executor grace periods will prevent payload execution	Low	Resolved
AG3-08	Payload IDs are not re-org safe	Low	Closed
AG3-09	Executor reconfiguration allows for execution of expired payloads	Informational	Resolved
AG3-10	Miscellaneous General Comments	Informational	Resolved

AG3-01	Payloads may not be created with the correct access level		
Asset	`PayloadsControllerCore.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

A payload consists of a sequence of actions. When it is created, the payload should record its highest access level. This level is used to determine security measures such as delay periods and also which executor is used. However, the level recorded is not the highest level, but the level of the last action in the sequence.

Payloads are initially created in `PayloadsControllerCore.createPayload()`. Early in this function, a new entry is created in the `_payloads` mapping and this is referenced by the `storage` variable `newPayload`. Because this is a new entry in the mapping, its `uint` values will default to zero.

```
70  uint40 payloadId = _payloadsCount++;
    Payload storage newPayload = _payloads[payloadId];
72  newPayload.creator = msg.sender;
    newPayload.state = PayloadState.Created;
74  newPayload.createdAt = uint40(block.timestamp);
    newPayload.ipfsHash = ipfsHash;
76
    PayloadsControllerUtils.AccessControl maximumAccessLevelRequired;
78  for (uint256 i = 0; i < actions.length; i++) {
      require(
80      _accessLevelToExecutorConfig[actions[i].accessLevel].executor !=
          address(0),
82      Errors.EXECUTOR_WAS_NOT_SPECIFIED_FOR_REQUESTED_ACCESS_LEVEL
      );
84
      newPayload.actions.push(actions[i]);
86
      if (actions[i].accessLevel > newPayload.maximumAccessLevelRequired) {
88      maximumAccessLevelRequired = actions[i].accessLevel;
      }
90  }
    newPayload.maximumAccessLevelRequired = maximumAccessLevelRequired;
```

Later, at line [89], within the block that is looping through the actions, there is a test to see if the current action's access level is higher than the maximum recorded access level for this payload. However, this test is using a different variable in its update on line [90] and its test on line [89]. The variable updated is the memory variable `maximumAccessLevelRequired`, created on line [79]. However, the variable tested is the storage reference `newPayload.maximumAccessLevelRequired`, which will be zero for each test, as it will not have been updated at this point in the execution.

Because the test on line [89] will always pass (given that zero is not a valid access level), `maximumAccessLevelRequired` will be set to the value of `actions[i].accessLevel` for the final value of `i`, and so the recorded maximum access level set on line [93] will be this value.

It is therefore possible for a payload to be created and voted on with a maximum access level of 1, whilst containing actions with an access level of 2. In `PayloadsControllerCore.executePayload()`, this value is used to check that all conditions of the highest security level have been satisfied before executing the actions. This means that the level 2 actions would be executed with only level 1 protections.

The contract's getter functions return values from `_payloads[payloadId]`, which would contain this incorrect value. Hence, the payload would be represented throughout the system as executing a lower access level than it does.

Recommendations

Change line [89] to check against the value of `maximumAccessLevelRequired` instead of `newPayload.maximumAccessLevelRequired`.

Resolution

The development team have implemented the recommendation on an internal commit.

AG3-02	Payloads are not validated during proposal creation		
Asset	`GovernanceCore.sol` & `PayloadsControllerCore.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

This issue was found and fixed independently during the course of the review.

Description

It is possible to create a proposal for a `payloadId` that does not exist. As a result, an attacker can trick the voter to support a malicious proposal. Additionally, a lack of validation implies the access level of the proposal is not verified.

The documentation states under the section High-level differences V2 vs V3 that *"before creation, the proposer or somebody else will need to deploy and register in an Aave contract the payload/s smart contract."* However, this logic is not implemented, and a proposer can create a proposal without creating the payload. As a result, it may be possible to trick `votingToken` holders to support a proposal that contain a malicious payload.

Additionally, there is no check that ensures that the payload access level matches with the proposal access level, so it is possible to execute a `Level_2` payload with a `Level_1` proposal.

One possible exploit scenario would be as follows:

1. The attacker creates a payload. The attacker is sure that most of the users will support this payload.
2. He creates the proposal with two payloads. The first payload is the one created in the first step. The second one is still not created. The second `payloadId` should be chosen carefully. The attacker may need to monitor the `PayloadsController` to figure out the number of payloads created in a period of time.
3. The attacker activates the voting of the created proposal.
4. As the first payload in the proposal created by the attacker is likely to be supported and as the second payload doesn't exist yet, the users may be tricked and vote for this proposal.
5. When the voting is over, and the proposal is queued, the attacker creates the malicious payload, executes the proposal and finally executes the payload. Note that the `payloadId` of the malicious payload should match the payload used when creating the proposal.

Such scenario is not likely to happen because a proposal with an empty payload should not be the voted on by users. Furthermore, the `Guardian` is able to intervene in such scenario to cancel the proposal.

Recommendations

Change the function `Governance._forwardPayloadForExecution()` such that it sends the proposal access level and the proposal creation time to the `PayloadController`. Then verify the match between the two access level and that the payload was created before the proposal.

Resolution

The development team resolved the issue on an internal commit.

The function `Governance._forwardPayloadForExecution()` now sends the proposal access level and the proposal voting activation time. Necessary changes have been also made to `PayloadsControllerCore._queuePayload()` to ensure the correct access level between payloads and proposal and to ensure that the voter will have all the information before voting and make their decisions by checking that the payload is created before the proposal voting activation time.

AG3-03	`PayloadController` is unable to receive native tokens		
Asset	`PayloadsController.sol`, `PayloadsControllerCore.sol`, `Executor.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

It is possible to register payloads that send native tokens, such as ETH, when executed. However, there is no practical way of sending native tokens to the `PayloadsController` contract.

Payload transactions are ultimately executed by the `Executor` contract, which cannot directly receive ETH transfers. The function `executeTransaction()` is payable, however it is only callable by the owner, `PayloadsController`. This means that `PayloadsController` is able to transfer ETH to `Executor`. However, `PayloadsController` does not have any payable functions.

It is possible to transfer ETH into `PayloadsController` and `Executor` through use of `selfdestruct` and through miner rewards. However, these methods are unreliable and unnecessarily awkward.

Recommendations

Add a `payable` function to `PayloadsController`. Whether it should be a general `payable fallback` or a specific named function is best chosen by the development team, in light of where and when they want ETH to enter the execution flow.

Resolution

The development team resolved the issue on an internal commit, which adds a `receive()` function in `PayloadsControllerCore` contract to enable receiving native tokens.

AG3-04	Removing all voting configs may lock the governance contract		
Asset	`GovernanceCore.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

This issue was found and fixed independently during the course of the review.

Description

In a situation where the governance contract controls the `GovernanceCore` ownership and all voting configs have `isActive` set to `false`, it would be impossible to process a vote to add a config.

The function to add voting configs, `setVotingConfigs()`, has the `onlyOwner` modifier. When the owner is a governance contract a vote must be executed to call ‘`setVotingConfigs()`. However, if all voting configs are disabled, it is not possible to execute a vote.

Recommendations

Consider adding a rescue function to allow the `guardian` to add a voting config if there are none remaining.

Resolution

The development team resolved the issue on an internal commit, which by removes the `isActive` flag from the voting configs.

AG3-05	The variable `_votingPortalsCount` is not updated correctly		
Asset	`GovernanceCore.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

This issue was found and fixed independently during the course of the review.

Description

There is a lack of checks in the function `_updateVotingPortals()`. As a result, the variable `_votingPortalsCount` may not contain the exact number of `VotingPortals`.

There are two cases where `_votingPortalsCount` will be incorrectly calculated:

- The variable `_votingPortalsCount` will incorrectly increment if the owner calls the function `addVotingPortals()` with a `VotingPortal` that has been already added.
- The variable `_votingPortalsCount` will incorrectly decrement if the owner calls the function `removeVotingPortals()` with a `VotingPortal` that has not been added.

The impact of this issue is the function `rescueVotingPortal()` may not work as intended. If `_votingPortalsCount` is understated then the guardian may call `rescueVotingPortal()` while there are still active portals. If `_votingPortalsCount` is overstated then the guardian may not be able to call `rescueVotingPortal()` although there are no active portals.

Recommendations

Add the following checks in the function `_updateVotingPortals()`. Check if the `VotingPortal` has not previously been added when incrementing `_votingPortalsCount`. Furthermore, check if the `VotingPortal` exists when decrementing `_votingPortalsCount`.

Resolution

The development team resolved the issue on an internal commit, which does not update the `_votingPortalsCount` if the `votingPortal` state is the same as the new state.

AG3-06	Voting duration is not checked before a proposal is queued or voted on		
Asset	`GovernanceCore.sol` & `VotingMachine.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

This issue was found and fixed independently during the course of the review.

Description

When successful votes are received for a proposal, it is queued for execution. The function that does this, `queueProposal()`, does not check that the voting duration of the proposal has passed.

The variable `proposal.votingDuration` is the duration of the vote. It is not validated to ensure sufficient time has passed.

This issue is mitigated by the voting machine, which receives the proposal's voting duration in `VotingMachine.receiveCrossChainMessage()` and validates the duration has elapsed before continuing the voting process.

Similarly, the function `voteViaPortal()` does not ensure that `proposal.votingDuration` has not elapsed. It is therefore possible to vote on a proposal after the duration has ended. Again this will be caught by the `VotingMachine` as the time will have expired and the cross chain message will revert.

Recommendations

Add a check to `queueProposal()` to ensure that `proposal.votingDuration` has passed since the vote was initiated.

Ensure `voteViaPortal()` occurs before the vote duration has elapsed.

Resolution

The development team resolved the issue on an internal commit, which updates the function `queueProposal()` such that it will reject proposals where the voting duration has passed.

The development team have decided against rejecting late votes in the `voteViaPortal()`. Due to the asynchronous clocks between chains it is possible for the vote to still be valid on the `VotingMachine`. Thus, votes are allowed to be sent and validity of the timestamp of the vote will be judged by the `VotingMachine`.

AG3-07	Small executor grace periods will prevent payload execution		
Asset	`PayloadsControllerCore.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The grace period of an executor config is how long, in seconds, the payload can be executed for. When executor configs are changed, there is a test to ensure that the grace period is not zero. However, there are other values that would or could render a payload impossible to execute.

A grace period of 1 would render the payload impossible to execute because the checks in `executePayload()` require the `block.timestamp` to be greater than `executionTime` and less than `executionTime + executorConfig.gracePeriod`. This is mathematically impossible if `executorConfig.gracePeriod` is 1.

Other small values might have a similar effect. If the code is running on a chain where blocks are mined more than one second apart, a short grace period can fall entirely between two blocks, making it impossible to have a valid execution time.

Also, an execution window of a single block would be highly vulnerable to disruption from network issues, node issues, or sudden spikes in gas fees delaying the execution transaction.

This issue is mitigated by the fact that executor configurations can be modified in `updateExecutors()`, although consider AG3-09.

Recommendations

Consider adding an immutable minimum grace period for executor configs, and test against this value on line [**221**], instead of testing that the value is not zero.

Resolution

The development team resolved the issue on an internal commit, which adds a new parameter `gracePeriod` to the struct `Payload` and assign to this parameter the new constant variable `GRACE_PERIOD` in the function `createPayload`. The grace period is now constant for all payloads, and it is not an executor parameter anymore.

AG3-08	Payload IDs are not re-org safe		
Asset	`PayloadControllerCore.sol` & `GovernanceCore.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Creation of payloads are not re-org safe and may result in a proposal with malicious payloads.

A payload is created via an incrementing nonce `_payloadsCount` in `createPayload()`. When there is a re-org and a new transaction which calls `createPayload()` is executed before the existing call, the new transaction will have the payload ID of the original transaction.

For example, consider two transactions which each call `createPayload()`, `transactionA` and `transactionB`. Say, `_payloadsCount` is 10. If `transactionA` is executed first it will have payload ID 10. However, if re-org occurs and now `transactionB` is executed first then it will have payload ID 10.

This is an issue with relation to `createProposal()` in the governance contract. If a proposal is created which points to payload ID 10, then there is a significant chain re-org on the `PayloadController` chain and a malicious user inserts a transaction which calls `createPayload()`. Now the malicious user's transaction will have payload ID 10.

This issue is rated as low severity due to the extended time in which the voting duration occurs. If a malicious payload is inserted into a vote then it is possible to cancel this proposal or payload via the guardian. Furthermore, there are delays between the vote finishing, queuing and execution which all allow for a payload to be cancelled.

Recommendations

Consider changing the payload ID to instead be a hash of the payload data and an incrementing nonce. In the case of a re-org the payload ID will change. Therefore, the proposal will instead point to an invalid payload which can not be executed.

Resolution

This issue has been acknowledged by the development team. The development team has clarified that there is no need to protect against the re-org as they have the social safeguards like guardians and time locks to protect against this vulnerability.

AG3-09	Executor reconfiguration allows for execution of expired payloads
Asset	`PayloadsControllerCore.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

It is possible to reconfigure an executor for an expired payload that is also queued such that it is executable.

Additionally, it is possible to reconfigure the executor for a payload whilst the payload is queued to block execution.

Both of these outcomes are produced by modifying the executor configs whilst a payload has been queued.

Consider the checks in `executePayload()` for execution time:

```
118 uint256 executionTime = payload.queuedAt + executorConfig.delay;
    require(block.timestamp > executionTime, Errors.TIMELOCK_NOT_FINISHED);
120 require(
      block.timestamp < executionTime + executorConfig.gracePeriod,
122   Errors.GRACE_PERIOD_FINISHED
    );
```

These checks are dependent on the values of `executorConfig`, which is an alias of `_accessLevelToExecutorConfig` and can be modified in `updateExecutors()`. Although `updateExecutors()` is only callable by `owner`, `executePayload()` is permissionless. Therefore, if an executor config were ever changed to significantly increase the value of `executorConfig.delay` or `executorConfig.gracePeriod`, any address could potentially call `executePayload()` and execute any payload previously assumed to be expired.

This risk is significantly mitigated as the guardian may cancel any payload which has not been executed.

If an executor config is modified after the payload has been queued, it may decrease the value of `executorConfig.delay` such that the check on line [**121**] becomes `false`. This would close the window for execution before it opened and prevent execution.

Recommendations

Consider storing the execution time and grace period for each payload such that changing configuration values do not impact existing payloads.

Resolution

The development team resolved the issue on an internal commit, which adds a new parameter `delay` to the struct `Payload` and assign a value this parameter in the function `createPayload`. As a result, the execution time now depends on the payload and not on the executor configuration and changing the configuration will not impact existing payloads.

AG3-10	Miscellaneous General Comments
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Typos and grammar**
   - `GovernanceCore.sol` line [**26**] "its" should be "it's" (for "it is").
   - `GovernanceCore.sol` line [**27**] "provably" should be "probably" (probably).
   - `GovernanceCore.sol` line [**237**] and `Errors.sol` the error constant `TO_MANY_TOKENS_FOR_VOTING` should be renamed `TOO_MANY_TOKENS_FOR_VOTING`.
   - `IVotingMachineWithProofs.sol` line [**323**] typo "hte".
   - `IPayloadsControllerCore.sol` line [**200**] the word "conform" here does not appear to make sense and should probably be "comprise" or "form".
2. **Error name**
   - In `VotingMachineWithProofs`, the error `PORTAL_VOTE_WITH_NO_VOTING_TOKENS` occurs if the number of tokens in the bridged tokens does not match the number of entries in the voting balance proof. The error name does not seem to reflect this and instead seems to imply there are no tokens in the bridged vote.
   - In the contract `GovernanceCore`, the function `queueProposal()` reverts with the error `CALLER_NOT_A_VALID_VOTING_PORTAL` when the proposal is not created yet. This error does not describe the correct reason behind the revert.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team have addressed all issue.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

```
test_processStorageRoot                                          PASSED  [1%]
test_processStorageSlot                                          PASSED  [2%]
test_getStorage                                                  PASSED  [3%]
test_executeTransaction                                          PASSED  [4%]
test_executeTransaction_selfdestruction                          PASSED  [6%]
test_basic                                                       PASSED  [7%]
test_constructor                                                 PASSED  [8%]
test_initialize                                                  PASSED  [9%]
test_create_proposal                                             PASSED  [11%]
test_create_proposal_without_payload                             PASSED  [12%]
test_create_proposal_not_approved_voting_portal                  PASSED  [13%]
test_create_proposal_proposition_power_low                       PASSED  [14%]
test_activate_voting                                             PASSED  [16%]
test_activate_voting_proposal_not_created_state                  PASSED  [17%]
test_activate_voting_cooldown_period_not_passed                  PASSED  [18%]
test_activate_voting_proposition_power_low                       PASSED  [19%]
test_vote_via_portal                                             PASSED  [20%]
test_vote_via_portal_proposal_not_active                         PASSED  [22%]
test_vote_via_portal_many_voting_tokens                          PASSED  [23%]
test_queue_proposal_passed_proposal                              PASSED  [24%]
test_queue_proposal_failed_proposal_case_1                       PASSED  [25%]
test_queue_proposal_failed_proposal_case_2                       PASSED  [27%]
test_queue_proposal_invalid_caller                               PASSED  [28%]
test_queue_proposal_not_active                                   PASSED  [29%]
test_execute_proposal                                            PASSED  [30%]
test_execute_proposal_not_in_queued_state                        PASSED  [32%]
test_execute_proposal_cooldown_period_not_passed                 PASSED  [33%]
test_execute_proposal_proposition_power_low                      PASSED  [34%]
test_cancel_proposal_case_1                                      PASSED  [35%]
test_cancel_proposal_case_2                                      PASSED  [37%]
test_cancel_proposal_wrong_proposal_state_case_1                 PASSED  [38%]
test_cancel_proposal_wrong_proposal_state_case_2                 PASSED  [39%]
test_add_voting_portals                                          PASSED  [40%]
test_remove_voting_portals                                       PASSED  [41%]
test_rescue_voting_portal                                        PASSED  [43%]
test_rescue_voting_portal_voting_count_not_zero                  PASSED  [44%]
test_set_power_strategy                                          PASSED  [45%]
test_set_voting_configs                                          PASSED  [46%]
test_constants_variable                                          PASSED  [48%]
test_get_voting_asset_list                                       PASSED  [49%]
test_get_voting_asset_config                                     PASSED  [50%]
test_basic                                                       PASSED  [51%]
test_constructor                                                 PASSED  [53%]
test_initialize                                                  PASSED  [54%]
test_updateExecutors                                             PASSED  [55%]
test_createPayload                                               PASSED  [56%]
test_createPayload_reverts                                       PASSED  [58%]
test_receiveCrossChainMessage                                    PASSED  [59%]
test_receiveCrossChainMessage_expired                            PASSED  [60%]
test_receiveCrossChainMessage_payload_created_after_proposal     PASSED  [61%]
test_cancelPayload                                               PASSED  [62%]
test_executePayload                                              PASSED  [64%]
test_submitVote_separate                                         PASSED  [65%]
test_getAccountSlotHash                                          PASSED  [66%]
test_constructor                                                 PASSED  [67%]
test_receiveCrossChainMessage_create_proposal                    PASSED  [69%]
test_receiveCrossChainMessage_bridge_vote                        PASSED  [70%]
test_receiveCrossChainMessage_encode_errors                      PASSED  [71%]
test_decodeVoteMessage                                           PASSED  [72%]
test_decodeProposalMessage                                       PASSED  [74%]
test_decodeMessage                                               PASSED  [75%]
test_submitVote                                                  PASSED  [76%]
```

```
test_submitVote_reverts                                    PASSED  [77%]
test_submitVoteBySignature                                 PASSED  [79%]
test_settleVoteFromPortal                                  PASSED  [80%]
test_closeAndSendVote                                      PASSED  [81%]
test_constructor                                           PASSED  [82%]
test_receive_cross_chain_message_delivered_vote_case_1     PASSED  [83%]
test_receive_cross_chain_message_delivered_vote_case_2     PASSED  [85%]
test_receive_cross_chain_message_not_delivered             PASSED  [86%]
test_forward_start_voting_message                          PASSED  [87%]
test_forward_start_voting_message_wrong_caller             PASSED  [88%]
test_forward_vote_message                                  PASSED  [90%]
test_forward_vote_message_voter_already_voted              PASSED  [91%]
test_constructor                                           PASSED  [92%]
test_getVotingAssetList                                    PASSED  [93%]
test_getVotingAssetConfig                                  PASSED  [95%]
test_hasRequiredRoots                                      PASSED  [96%]
test_getVotingPower_aave                                   PASSED  [97%]
test_getVotingPower_stkaave                                PASSED  [98%]
test_getVotingPower_a_aave                                 PASSED  [100%]
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

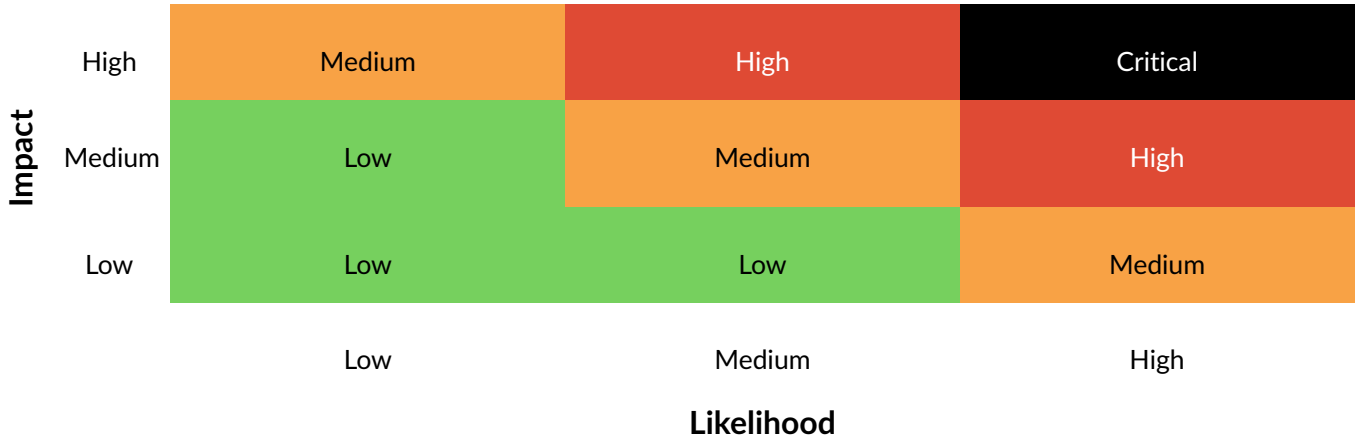

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].