SMART CONTRACT AUDIT REPORT

for

AAVE

Prepared By: Shuxiao Wang

Hangzhou, China
December 10, 2020

Document Properties

Client	Aave
Title	Smart Contract Audit Report
Target	GovernanceV2 and AAVE Token
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 10, 2020	Xuxian Jiang	Final Release
1.0-rc	December 7, 2020	Xuxian Jiang	Release Candidate
0.2	December 6, 2020	Xuxian Jiang	Additional Findings
0.1	December 3, 2020	Xuxian Jiang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of `Aave`'s **GovernanceV2 and AAVE Token**, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of `governanceV2` and `AAVE` token contract can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Aave

`Aave` is a decentralized non-custodial money market protocol where users can participate as depositors or borrowers. Depositors provide liquidity to the market to earn a passive income, while borrowers are able to borrow in an over-collateralized (perpetually) or under-collateralized (one-block liquidity) fashion. The recent upgrade of `AaveV2` not only addresses some of the suboptimal solutions implemented in V1 (e.g., by allowing for `AToken` upgradeability and simplified overall architecture), but also provides additional features, e.g., debt tokenization, collateral trading, and new flashloans. This audit covers the new `governanceV2` and the governance-compatible `AAVE` token contracts.

The basic information of `Aave`'s `governanceV2` and `AAVE` token is as follows:

Table 1.1: Basic Information of Aave's GovernanceV2 and AAVE Token

Item	Description
Issuer	Aave
Website	http://aave.com/
Audit Modules	GovernanceV2 and AAVE Token
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 10, 2020

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit.

- https://github.com/aave/aave-token.git (9f384f)
- https://github.com/aave/governance-v2.git (8ac66e5)

And here are the final commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/aave/aave-token-v2.git (6ebf51d)
- https://github.com/aave/governance-v2.git (7b0e1e2)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

PeckShield

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the Aave's governance subsystem and its new token contract. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	3	■■■
Informational	2	■■
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 low-severity vulnerabilities and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Removal of Unused Code	Coding Practices	Fixed
PVE-002	Informational	Improved Gas Optimization in _moveDelegates()	Coding Practices	Fixed
PVE-003	Low	Additional Validation When Canceling Proposals	Business Logics	Fixed
PVE-004	Low	Corner Case Handling in isQuorumValid() And isVoteDifferentialValid()	Business Logic	Fixed
PVE-005	Low	Improved DelegatedCall Execution in executeTransaction()	Business Logic	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Removal of Unused Code

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GovernancePowerDelegationERC20`
- Category: Coding Practices [3]
- CWE subcategory: CWE-563 [1]

Description

The `Aave`'s `governance` subsystem and `AAVE` token contract make good use of a number of reference contracts, such as `ERC20`, `SafeMath`, `VersionedInitializable`, and `Ownable`, to facilitate its code implementation and organization. For example, the `AaveGovernanceV2` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `GovernancePowerDelegationERC20` contract, there is a routine, i.e., `getPowerAtBlock()`, to query the delegated power of a user at a certain block. To elaborate, we show its code snippet below. Within this particular routine, there is a line of code (line 95) that is redundant and can be safely removed.

```
80  /**
81   * @dev returns the delegated power of a user at a certain block
82   * @param user the user
83   **/
84  function getPowerAtBlock(
85    address user,
86    uint256 blockNumber,
87    DelegationType delegationType
88  ) external override view returns (uint256) {
89    (
90      mapping(address => mapping(uint256 => Snapshot)) storage snapshots,
91      mapping(address => uint256) storage snapshotsCounts,
92
```

```
    ) = _getDelegationDataByType(delegationType);

    uint256 snapshotsCount = snapshotsCounts[user];

    return _searchByBlockNumber(snapshots, snapshotsCounts, user, blockNumber);
  }
```

Listing 3.1: GovernancePowerDelegationERC20::getPowerAtBlock()

Recommendation Consider the removal of the unused code.

Status The issue has been fixed in this comment: `4df3e3f`.

3.2 Improved Gas Optimization in _moveDelegates()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GovernancePowerDelegationERC20`
- Category: Coding Practices [3]
- CWE subcategory: CWE-563 [1]

Description

The `Aave`'s `AAVE` token contract has been enhanced to be used for voting and governance. The key enhancement is implemented by a helper routine named `_moveDelegates()` for delegating the voting/proposition power from one user to another. To elaborate, we show below its implementation.

The logic is rather straightforward in firstly retrieving relevant delegation data from the delegation type, then processing the `from` user from which delegated power is moved, and next handling the `to` user that will receive the delegated power. Each handling may update internal account-specific snapshots (via `writeSnapshot()`).

During our analysis, we observe the opportunity to further optimize the gas usage. For example, when processing the `from` user from which delegated power is moved, the user's current `snapshotsCounts` have been accessed twice (lines 151 and 154). The second storage read via `SLOAD` at line 154 is unnecessary, i.e., `previous = snapshots[from][snapshotsCounts[from].sub(1)].value`. For better gas efficiency, we can optimize the statement as `previous = snapshots[from][fromSnapshotsCount -1].value`.

```
126  /**
127   * @dev moves delegated power from one user to another
128   * @param from the user from which delegated power is moved
129   * @param to the user that will receive the delegated power
130   * @param amount the amount of delegated power to be moved
131   * @param delegationType the type of delegation (VOTING_POWER, PROPOSITION_POWER)
```

```
  **/
function _moveDelegates(
  address from,
  address to,
  uint256 amount,
  DelegationType delegationType
) internal {
  if (from == to) {
    return;
  }

  (
    mapping(address => mapping(uint256 => Snapshot)) storage snapshots,
    mapping(address => uint256) storage snapshotsCounts,

  ) = _getDelegationDataByType(delegationType);

  if (from != address(0)) {
    uint256 previous = 0;
    uint256 fromSnapshotsCount = snapshotsCounts[from];

    if (fromSnapshotsCount != 0) {
      previous = snapshots[from][snapshotsCounts[from].sub(1)].value;
    } else {
      previous = balanceOf(from);
    }

    _writeSnapshot(
      snapshots,
      snapshotsCounts,
      from,
      uint128(previous),
      uint128(previous.sub(amount))
    );

    emit DelegatedPowerChanged(from, previous.sub(amount), delegationType);
  }
  if (to != address(0)) {
    uint256 previous = 0;
    uint256 toSnapshotsCount = snapshotsCounts[to];
    if (toSnapshotsCount != 0) {
      previous = snapshots[to][snapshotsCounts[to].sub(1)].value;
    } else {
      previous = balanceOf(to);
    }

    _writeSnapshot(
      snapshots,
      snapshotsCounts,
      to,
      uint128(previous),
      uint128(previous.add(amount))
```

```
184       );
185
186       emit DelegatedPowerChanged(to, previous.add(amount), delegationType);
187     }
188   }
```

Listing 3.2: GovernancePowerDelegationERC20::_moveDelegates()

Note the handling of the `to` user that will receive the delegated power has a similar issue. The second storage read via `SLOAD` at line 173 is unnecessary, i.e., `previous = snapshots[to][snapshotsCounts[to].sub(1)].value`. For better gas efficiency, we can optimize the statement as `previous = snapshots[to][toSnapshotsCount-1].value`.

Recommendation Optimize the gas efficiency by avoiding unnecessary storage reads.

Status The issue has been fixed in this comment: `4df3e3f`.

3.3 Additional Validation When Canceling Proposals

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AaveGovernanceV2`
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [2]

Description

According to the governance design, a proposal may fall in the following state, i.e., `Pending`, `Canceled`, `Active`, `Succeeded`, `Failed`, `Queued`, `Executed`, and `Expired`. The transition from one state to another is defined by the state transition rule hardcoded in the governance subsystem.

In the following, we examine a particular `Canceled` state that can be entered if the proposal is canceled (via the `cancel()` routine). To illustrate, we show the routine's implementation. A proposal can be canceled if the proposer has lost the proposition power to be below the required threshold. Also, if a proposal is already canceled or executed, the attempt to `cancel()` is reverted.

```
  /**
   * @dev Cancels a Proposal.
   * - Callable by the _guardian with relaxed conditions, or by anybody if the
       conditions of
   *   cancellation on the executor are fulfilled
   * @param proposalId id of the proposal
   **/
  function cancel(uint256 proposalId) external override {
    ProposalState state = getProposalState(proposalId);
    require(
```

```
      state != ProposalState.Executed && state != ProposalState.Canceled,
      'ONLY_BEFORE_EXECUTED'
    );

    Proposal storage proposal = _proposals[proposalId];
    require(
      msg.sender == _guardian
        IProposalValidator(address(proposal.executor)).validateProposalCancellation(
          this,
          proposal.creator,
          block.number - 1
        ),
      'PROPOSITION_CANCELLATION_INVALID'
    );
    proposal.canceled = true;
    for (uint256 i = 0; i < proposal.targets.length; i++) {
      proposal.executor.cancelTransaction(
        proposal.targets[i],
        proposal.values[i],
        proposal.signatures[i],
        proposal.calldatas[i],
        proposal.executionTime,
        proposal.withDelegatecalls[i]
      );
    }

    emit ProposalCanceled(proposalId);
  }
```

Listing 3.3: AaveGovernanceV2::cancel()

We note that if a proposal has been expired, there is also no need to cancel it.

Recommendation Revise the `cancel()` logic by reverting the transaction as well if current proposal is expired.

Status The issue has been fixed in this comment: `a085a95`.

3.4 Corner Case Handling in isQuorumValid() And isVoteDifferentialValid()

- ID: PVE-004
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `ProposalValidator`
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [2]

Description

The governance subsystem in Aave specifies the entire life-cycle of a proposal. A proposal, if successfully passed, will lead to its activation in triggering the enclosed `executor`.

To elaborate, we show below the code snippet of the `isProposalPassed()` routine that is defined to assess whether a proposal has been passed or not. For a proposal to pass, it requires two conditions: the first one (line 134) is the proposal has reached quorum in receiving enough `FOR` votes. Note that the quorum is not to count the number of votes reached, but the number of `FOR` votes reached. The second one (line 135) is the proposal has enough extra `FOR` votes than `AGAINST` votes by at least `VOTE_DIFFERENTIAL` percentage of total supply.

```
122   /**
123    * @dev Returns whether a proposal passed or not
124    * @param governance Governance Contract
125    * @param proposalId Id of the proposal to set
126    * @return true if proposal passed
127    **/
128   function isProposalPassed(IAaveGovernanceV2 governance, uint256 proposalId)
129     external
130     view
131     override
132     returns (bool)
133   {
134     return (isQuorumValid(governance, proposalId) &&
135       isVoteDifferentialValid(governance, proposalId));
136   }
```

Listing 3.4: ProposalValidator :: isProposalPassed()

In the following, we further show the implementation of two routines, i.e., `isQuorumValid()` and `isVoteDifferentialValid()`. These two routine validate whether a proposal meets the above two conditions, respectively. In `isQuorumValid()`, we notice that it validates the following: `proposal.forVotes > getMinimumVotingPowerNeeded(votingSupply)`. If we consider a corner case where `proposal.forVotes == getMinimumVotingPowerNeeded(votingSupply)`, the proposal will not be considered pass, even though the received `FOR` has reached the defined quorum.

```
/**
 * @dev Check whether a proposal has reached quorum, ie has enough FOR-voting-power
 * Here quorum is not to understand as number of votes reached, but number of for-votes reached
 * @param governance Governance Contract
 * @param proposalId Id of the proposal to verify
 * @return voting power needed for a proposal to pass
 **/
function isQuorumValid(IAaveGovernanceV2 governance, uint256 proposalId)
  public
  view
  override
  returns (bool)
{
  IAaveGovernanceV2.ProposalWithoutVotes memory proposal = governance.getProposalById(proposalId);
  uint256 votingSupply = IGovernanceStrategy(proposal.strategy).getTotalVotingSupplyAt(
    proposal.startBlock
  );

  return proposal.forVotes > getMinimumVotingPowerNeeded(votingSupply);
}
```

Listing 3.5: ProposalValidator :: isQuorumValid()

Similarly, our analysis of `isVoteDifferentialValid()` shows a similar issue in missing the corner case (lines 191 – 194).

```
173 /**
174  * @dev Check whether a proposal has enough extra FOR-votes than AGAINST-votes
175  * FOR VOTES - AGAINST VOTES > VOTE_DIFFERENTIAL * voting supply
176  * @param governance Governance Contract
177  * @param proposalId Id of the proposal to verify
178  * @return true if enough For-Votes
179  **/
180 function isVoteDifferentialValid(IAaveGovernanceV2 governance, uint256 proposalId)
181   public
182   view
183   override
184   returns (bool)
185 {
186   IAaveGovernanceV2.ProposalWithoutVotes memory proposal = governance.getProposalById(proposalId);
187   uint256 votingSupply = IGovernanceStrategy(proposal.strategy).getTotalVotingSupplyAt(
188     proposal.startBlock
189   );
190
191   return (proposal.forVotes.mul(ONE_HUNDRED_WITH_PRECISION).div(votingSupply) >
192     proposal.againstVotes.mul(ONE_HUNDRED_WITH_PRECISION).div(votingSupply).add(
193       VOTE_DIFFERENTIAL
194     ));
```

```
}
```

Listing 3.6: ProposalValidator :: isVoteDifferentialValid ()

Recommendation Revise both `isQuorumValid()` and `isVoteDifferentialValid()` to accommodate the above corner cases.

Status The issue has been fixed in this comment: `384a149`.

3.5 Improved DelegatedCall Execution in executeTransaction()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ExecutorWithTimelock`
- Category: Business Logics [4]
- CWE subcategory: CWE-841 [2]

Description

As mentioned in Section 3.4, the governance subsystem in Aave specifies the entire life-cycle of a proposal. A proposal, if successfully passed, will lead to its activation in triggering the enclosed `executor`. In Section 3.4, we have examined the two requirements for a proposal to considered pass and make it ready for execution.

In this section, we examine the proposal execution logic. Accordingly, we show below the `execute()` routine in `AaveGovernanceV2`. After validating the proposal state, the execution logic is relayed to the `executor` (line 217).

```
208 /**
209  * @dev Execute the proposal (If Proposal Queued)
210  * @param proposalId id of the proposal to execute
211  **/
212 function execute(uint256 proposalId) external payable override {
213   require(getProposalState(proposalId) == ProposalState.Queued, 'ONLY_QUEUED_PROPOSALS
        ');
214   Proposal storage proposal = _proposals[proposalId];
215   proposal.executed = true;
216   for (uint256 i = 0; i < proposal.targets.length; i++) {
217     proposal.executor.executeTransaction{value: proposal.values[i]}(
218       proposal.targets[i],
219       proposal.values[i],
220       proposal.signatures[i],
221       proposal.calldatas[i],
222       proposal.executionTime,
223       proposal.withDelegatecalls[i]
224     );
```

```
    }
    emit ProposalExecuted(proposalId, msg.sender);
  }
```

Listing 3.7: AaveGovernanceV2::execute()

Inside `executor`, the `executeTransaction()` routine handles the actual proposal execution. We notice that the proposal execution supports both `delegatecall` (line 207) and normal call (line 210). As this routine is marked as `payable` and the proposal execution may require certain amount of `Ether` as the payment, there is a needed to `require(msg.value >= value)` for the `delegatecall` case. Note the normal call specifies $target.call\{value : value\}(callData)$ (line 210), which reverts if there is an insufficient balance (including this payment).

```
168   /**
169    * @dev Function, called by Governance, that cancels a transaction, returns the callData executed
170    * @param target smart contract target
171    * @param value wei value of the transaction
172    * @param signature function signature of the transaction
173    * @param data function arguments of the transaction or callData if signature empty
174    * @param executionTime time at which to execute the transaction
175    * @param withDelegatecall boolean, true = transaction delegatecalls the target, else calls the target
176    * @return the callData executed as memory bytes
177    **/
178   function executeTransaction(
179     address target,
180     uint256 value,
181     string memory signature,
182     bytes memory data,
183     uint256 executionTime,
184     bool withDelegatecall
185   ) public payable override onlyAdmin returns (bytes memory) {
186     bytes32 actionHash = keccak256(
187       abi.encode(target, value, signature, data, executionTime, withDelegatecall)
188     );
189     require(_queuedTransactions[actionHash], 'ACTION_NOT_QUEUED');
190     require(block.timestamp >= executionTime, 'TIMELOCK_NOT_FINISHED');
191     require(block.timestamp <= executionTime.add(GRACE_PERIOD), 'GRACE_PERIOD_FINISHED');
192
193     _queuedTransactions[actionHash] = false;
194
195     bytes memory callData;
196
197     if (bytes(signature).length == 0) {
198       callData = data;
199     } else {
200       callData = abi.encodePacked(bytes4(keccak256(bytes(signature))), data);
201     }
202
```

```
    bool success;
    bytes memory resultData;
    if (withDelegatecall) {
      // solium-disable-next-line security/no-call-value
      (success, resultData) = target.delegatecall(callData);
    } else {
      // solium-disable-next-line security/no-call-value
      (success, resultData) = target.call{value: value}(callData);
    }

    require(success, 'FAILED_ACTION_EXECUTION');

    emit ExecutedAction(
      actionHash,
      target,
      value,
      signature,
      data,
      executionTime,
      withDelegatecall,
      resultData
    );

    return resultData;
  }
```

Listing 3.8: ExecutorWithTimelock::executeTransaction()

Recommendation Add the `Ether` payment requirement in the `delegatecall` scenario.

Status The issue has been fixed in this comment: `0dbeabe`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Aave` protocol's `governanceV2` subsystem and the protocol-wide `AAVE` token contract. The system presents a unique offering in decentralized non-custodial money market protocol where users can participate as depositors or borrowers. We are impressed by the overall design and implementation. The current code base is well organized and those identified issues are promptly confirmed and fixed.

As a final precaution, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

PeckShield

References

[1] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[3] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[4] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[5] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[6] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[7] PeckShield. PeckShield Inc. https://www.peckshield.com.