



SMART CONTRACT AUDIT REPORT

for

dTrade



Prepared By: Yiqun Chen

PeckShield
July 2, 2021

Document Properties

Client	dTrade
Title	Smart Contract Audit Report
Target	dTrade
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 2, 2021	Xuxian Jiang	Final Release
1.0-rc1	June 5, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About dTrade	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Inconsistency Between Document and Implementation	11
3.2	Suggested Reentrancy Prevention in TokenVesting And InsuranceFund	14
3.3	Improved Handling of Corner Cases in Proposal Submission	15
3.4	Full Charge of Proposal Execution Cost From Accompanying msg.value	18
4	Conclusion	20
	References	21

1 | Introduction

Given the opportunity to review the **dTrade** design document and related smart contract source code, 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 smart contracts 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 dTrade

The dTrade contracts for audit include four main components: dTrade Exchange Token (DET), Insurance Mining, Vesting, and Governance. The DET is the ERC20-compliant governance token with the total supply of 200,000,000 DET emitted over a 10-year period. Insurance Mining allows users to lock USDC in an insurance fund and receive DET rewards each block. Vesting enables the distribution of tokens to investors, advisors and team over a vesting period. (The vesting contract also allows locked tokens to be used for voting and staking.) Governance is a DAO and all proposals for upgrades and parameter modifications will take place through governance.

The basic information of dTrade is as follows:

Table 1.1: Basic Information of dTrade

Item	Description
Issuer	dTrade
Type	Polkadot Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 2, 2021

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

- https://github.com/dtradeorg/dtrade_v1.git (8c6e12b)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/dtradeorg/dtrade_v1.git (92c3804)

1.2 About PeckShield

PeckShield Inc. [10] 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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [9]:

- 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.

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

To evaluate the risk, we go through a checklist of 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) [8], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, 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 ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.



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 Logic	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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the dTrade protocol. 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 logic, 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	1	
Total	4	

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 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 1 informational recommendation.

Table 2.1: Key dTrade Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Inconsistency Between Document and Implementation	Coding Practices	Resolved
PVE-002	Low	Suggested Reentrancy Prevention in TokenVesting And InsuranceFund	Time and State	Resolved
PVE-003	Low	Improved Handling of Corner Cases in Proposal Submission	Business Logic	Resolved
PVE-004	Low	Full Charge of Proposal Execution Cost From Accompanying msg.value	Business Logic	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Inconsistency Between Document and Implementation

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

There are a few misleading comments embedded among lines of solidity code, which bring unnecessary hurdles to understand and/or maintain the software.

A few example comments can be found at line 86 of the `VestedBalance` data structure in the `TokenVesting` contract, line 14 in the `InsuranceFund` contract, line 301 in the `InsuranceFund::lock()` function, and line 572 of the `InsuranceFund::_computeUpdatedMarketState()` function. Using the `VestedBalance` data structure as an example, the preceding description indicates that “If the vesting duration is 30 days with a cliff of 2 days then the total number of sessions is 15.” However, the total number of sessions needs to be fully divisible by 4.

```
73  /// @notice struct for storing the vested balance of a user
74  /// @param exists a boolean to show if a user exists or not
75  /// @param provider address of the controller which can revoke the allocation of
    coins
76  /// @param revocable if true then provider can halt/stop vesting
77  /// @param start vesting period starting block timestamp
78  /// @param duration duration of vesting period in seconds
79  /// @param cliff interval after which coins will get unlocked in seconds
80  /// @param totalBalance total number of DET's to be given to user over the period of
    vesting
81  /// @param balanceClaimed amount of balance that user has claimed
82  /// @param fixedInterval if fixedInterval then same amount of tokens are going to
    get
83  /// unlocked after every cliff else 40% in first quarter,
84  /// 30% in 2nd, 20% in 3rd and 10% last quarter of duration
```

```

85     /// @param totalSessions total number of sessions, the vesting period has. If the
      vesting duration is 30 days
86     /// with a cliff of 2 days then the total number of sessions is 15.
87     /// @param sessionsClaimed number of sessions/periods of vesting that user has
      claimed. If the total sessions the vesting
88     /// period has are 30 and user has claimed released tokens for 5 periods, that will
      be the number of sessionsClaimed
89     /// @param inProgress true if the vesting period is still on going, false other wise
      . Its set to false in 2 cases:
90     /// i) the vesting period has expired ii) the provider has revoked the vesting coins
91     struct VestedBalance {
92         bool exists;
93         address provider;
94         bool revocable;
95         uint256 start;
96         uint256 duration;
97         uint256 cliff;
98         Decimal.decimal totalBalance;
99         Decimal.decimal balanceClaimed;
100        bool fixedInterval;
101        uint256 totalSessions;
102        uint256 sessionsClaimed;
103        bool inProgress;
104    }

```

Listing 3.1: TokenVesting::VestedBalance

Moreover, according to the design document, the InsuranceFund contract allows users to lock USDC in an insurance fund and receive DET rewards each block. However, as shown in the following lock() function, it indicates the operation to “update locked USDT in system” (line 301), not USDC.

```

264     function lock(
265         Decimal.decimal calldata _amount,
266         Decimal.decimal calldata _period
267     ) public {
268         require(
269             _amount.toUint() != 0,
270             "InsuranceFund::lock: Input amount is zero"
271         );
272
273         address _user = _msgSender();
274         InsuranceProvider storage provider = insuranceProviders[_user];
275
276         // compute coin weightage based on the number of time the coins are locked for
277         (Decimal.decimal memory coinWeightage, Decimal.decimal memory factor) =
278             computeCoinWeight(_amount, _period);
279
280         if (!provider.exists) {
281             provider.exists = true;
282         }
283
284         // claim rewards till current block based on current system state

```

```

285 // and users propotion in the system
286 _claimRewardsForUser(_user);
287
288 // update user coinsLocked
289 provider.balance.coinsLocked = provider.balance.coinsLocked.addD(
290     _amount
291 );
292
293 // update user coinsWeight
294 provider.balance.coinsWeight = provider.balance.coinsWeight.addD(
295     coinWeightage
296 );
297
298 // update balance block number
299 provider.balance.blockNumber = _blockNumber();
300
301 // update locked USDT in system
302 _updateLockedBalance(Action.STAKED, _amount, coinWeightage);
303
304 // update the trial of locked balance for user by adding an instance
305 // at the end of the trail map
306 provider.balanceTrail[provider.balanceTrailMapLen] = BalanceTrail({
307     lockedAtTimeStamp: _blockTimeStamp(),
308     unlockAfterTimeStamp: _computeUnlockTime(_period),
309     weightFactor: factor,
310     coinsLocked: _amount,
311     coinsWeight: coinWeightage
312 });
313 provider.balanceTrailMapLen = provider.balanceTrailMapLen.add(1);
314
315 // make the transfer
316 _transferFrom(IERC20(insuranceToken), _user, address(this), _amount);
317
318 emit InsuranceChanged(Action.STAKED, _user, _amount);
319 }

```

Listing 3.2: InsuranceFund::lock()

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status The issue has been fixed by this commit: 92c3804.

3.2 Suggested Reentrancy Prevention in TokenVesting And InsuranceFund

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Time and State [7]
- CWE subcategory: CWE-663 [2]

Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [12] exploit, and the recent Uniswap/Lendf.Me hack [11].

We notice there are a number of occasion where the checks-effects-interactions principle is violated. Using the Buyer as an example, the deposit() function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy.

Apparently, the interaction with the external contract (line 336) starts before effecting the update on internal states (lines 339, 342, and 345), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching re-entrancy via the same entry function.

```

318     function claimReleasedTokens(address _provider) public {
319         address user = _msgSender();
320         uint256 timestamp = _blockTimestamp();
321
322         // check if caller has any claimable tokens
323         require(
324             hasReleasedTokens(user, _provider, timestamp),
325             "TokenVesting::claimReleasedTokens: No releasable tokens at this moment"
326         );
327
328         // get caller balance
329         VestedBalance storage balance = vestings[user].vestedBalance[_provider];
330
331         // compute the amount claimable at current timestamp
332         (Decimal.decimal memory amount, uint256 sessionsClaimed) =
333             (_calcClaimableTokens(user, _provider, timestamp));

```

```

334
335     // transfer the dets from contract to user
336     _transfer(detToken, user, amount);
337
338     // update sessions claimed
339     balance.sessionsClaimed = sessionsClaimed;
340
341     // update the total amount of balance claimed
342     balance.balanceClaimed = balance.balanceClaimed.addD(amount);
343
344     // set progress to false if all the amount has been claimed
345     balance.inProgress = balance.sessionsClaimed < balance.totalSessions;
346
347     emit ClaimedReleasedTokens(user, _provider, amount, timestamp);
348 }

```

Listing 3.3: TokenVesting::claimReleasedTokens()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy. However, it is important to take precautions in making use of `nonReentrant` to block possible re-entrancy. Note similar issues exist in other functions, including `TokenVesting::vestTokens()`/`revokeVesting()`, and `InsuranceFund::claimAccruedRewards()`/`lock()`/`unlock()`.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible re-entrancy.

Status The issue has been fixed by this commit: [92c3804](#).

3.3 Improved Handling of Corner Cases in Proposal Submission

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Governance
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [4]

Description

The dTrade protocol adopts the governance implementation from Compound by accordingly adjusting its governance token and related parameters, e.g., `quorumVotes` and `proposalThreshold`. In the following, we elaborate one corner case during a proposal submission, especially regarding the proposer qualification.

To be qualified as a proposer, the governance subsystem requires the proposer to obtain a sufficient number of votes, including from the proposer herself and other voters. The threshold is specified

by proposalThreshold. In dTrade, this number will be initialized through the customized initialize() routine, which ensures the threshold falls between [MIN_PROPOSAL_THRESHOLD, MAX_PROPOSAL_THRESHOLD].

```

203     function propose(address[] memory targets, uint[] memory values, string[] memory
        signatures, bytes[] memory calldatas, string memory description) public returns
        (uint) {
204         // Reject proposals before initiating as Governor
205         require(det.getPriorVotes(msg.sender, sub256(block.number, 1)) >
            proposalThreshold, "DTrade Governance::propose: proposer votes below
            proposal threshold");
206         require(targets.length == values.length && targets.length == signatures.length
            && targets.length == calldatas.length, "DTrade Governance::propose: proposal
            function information arity mismatch");
207         require(targets.length != 0, "DTrade Governance::propose: must provide actions")
            ;
208         require(targets.length <= proposalMaxOperations, "DTrade Governance::propose:
            too many actions");

210         uint latestProposalId = latestProposalIds[msg.sender];
211         if (latestProposalId != 0) {
212             ProposalState proposersLatestProposalState = state(latestProposalId);
213             require(proposersLatestProposalState != ProposalState.Active, "DTrade
                Governance::propose: one live proposal per proposer, found an already
                active proposal");
214             require(proposersLatestProposalState != ProposalState.Pending, "DTrade
                Governance::propose: one live proposal per proposer, found an already
                pending proposal");
215         }

217         uint startBlock = add256(block.number, votingDelay);
218         uint endBlock = add256(startBlock, votingPeriod);

220         proposalCount++;
221         Proposal storage newProposal = proposals[proposalCount];
222         newProposal.id = proposalCount;
223         newProposal.proposer = msg.sender;
224         newProposal.eta = 0;
225         newProposal.targets = targets;
226         newProposal.values = values;
227         newProposal.signatures = signatures;
228         newProposal.calldatas = calldatas;
229         newProposal.startBlock = startBlock;
230         newProposal.endBlock = endBlock;
231         newProposal.forVotes = 0;
232         newProposal.againstVotes = 0;
233         newProposal.abstainVotes = 0;
234         newProposal.canceled = false;
235         newProposal.executed = false;

237         latestProposalIds[newProposal.proposer] = newProposal.id;

239         emit ProposalCreated(newProposal.id, msg.sender, targets, values, signatures,

```



```

240         calldatas, startBlock, endBlock, description);
241     return newProposal.id;
    }

```

Listing 3.4: Governance::propose()

If we examine the `propose()` logic, when a proposal is being submitted, the governance verifies up-front the qualification of the proposer (line 205): `require(det.getPriorVotes(msg.sender, sub256(block.number, 1)) > proposalThreshold)`. Note that the number of prior votes is strictly higher than `proposalThreshold`.

However, if we check the proposal cancellation logic, i.e., the `cancel()` function, a proposal can be canceled (line 285) if the number of prior votes (before current block) is strictly smaller than `proposalThreshold`. The corner case of having an exact number prior votes as the threshold, though unlikely, is largely unattended. It is suggested to accommodate this particular corner case as well.

```

281     function cancel(uint proposalId) external {
282         require(state(proposalId) != ProposalState.Executed, "DTrade Governance::cancel:
            cannot cancel executed proposal");

284         Proposal storage proposal = proposals[proposalId];
285         require(msg.sender == proposal.proposer || det.getPriorVotes(proposal.proposer,
            sub256(block.number, 1)) < proposalThreshold, "DTrade Governance::cancel:
            proposer above threshold");

287         proposal.canceled = true;
288         for (uint i = 0; i < proposal.targets.length; i++) {
289             timelock.cancelTransaction(proposal.targets[i], proposal.values[i], proposal
                .signatures[i], proposal.calldatas[i], proposal.eta);
290         }

292         emit ProposalCanceled(proposalId);
293     }

```

Listing 3.5: Governance::cancel()

Recommendation Accommodate the corner case by also allowing the proposal to be successfully submitted when the number of proposer's prior votes is exactly the same as the required threshold, i.e., `proposalThreshold`.

Status The issue has been fixed by this commit: 92c3804.

3.4 Full Charge of Proposal Execution Cost From Accompanying msg.value

- ID: PVE-013
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Governance
- Category: Business Logic [6]
- CWE subcategory: CWE-770 [3]

Description

The dTrade protocol adopts the governance implementation from Compound by adjusting its governance token and related parameters, e.g., `quorumVotes` and `proposalThreshold`. The original governance has been successfully audited by OpenZeppelin.

In the following, we would like to comment on a particular issue regarding the proposal execution cost. Note that the actual proposal execution is kicked off by invoking the governance's `execute()` function. This function is marked as `payable`, indicating the transaction sender is responsible for supplying required amount of native tokens as each inherent action (line 271) in the proposal may require accompanying certain native tokens, specified in `proposal.values[i]`, where i is the i^{th} action inside the proposal.

```

262  /**
263   * @notice Executes a queued proposal if eta has passed
264   * @param proposalId The id of the proposal to execute
265   */
266  function execute(uint proposalId) external payable {
267      require(state(proposalId) == ProposalState.Queued, "DTrade Governance::execute:
268          proposal can only be executed if it is queued");
269      Proposal storage proposal = proposals[proposalId];
270      proposal.executed = true;
271      for (uint i = 0; i < proposal.targets.length; i++) {
272          timelock.executeTransaction{value: proposal.values[i]}(proposal.targets[i],
273              proposal.values[i], proposal.signatures[i], proposal.calldatas[i],
274              proposal.eta);
275      }
276      emit ProposalExecuted(proposalId);
277  }

```

Listing 3.6: Governance::execute()

Though it is likely the case that a majority of these actions do not require any native token i.e., `proposal.values[i] = 0`, we may be less concerned on the payment of required tokens for the proposal execution. However, in the unlikely case of certain particular actions that do need native tokens the issue of properly attributing the associated cost arises. With that, we need to better keep

track of native tokens charged for each action and ensure that the transaction sender (who initiates the proposal execution) actually pays the cost. In other words, we do not rely on the governance's balance of native tokens for the payment. Another alternative is to implement the `receive()` function inside the `Governance` contract.

Recommendation Properly charge the proposal execution cost by ensuring the amount of accompanying tokens deposit is sufficient. If necessary, we can also return possible leftover, if any, back to the sender.

Status This issue has been confirmed. Considering the possibility that the `TimeLock` contract, if necessary, should be able to retrieve funds in a subsequent transaction, the team decides no change necessary for the time being.



4 | Conclusion

In this audit, we have analyzed the dTrade design and implementation. The system presents a unique, robust offering as a decentralized non-custodial protocol with the support for governance, insurance mining, and vesting. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, we need to emphasize that [Solidity](#)-based 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.



References

- [1] MITRE. CWE-1041: Use of Redundant Code. <https://cwe.mitre.org/data/definitions/1041.html>.
- [2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.html>.
- [3] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. <https://cwe.mitre.org/data/definitions/770.html>.
- [4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. <https://cwe.mitre.org/data/definitions/837.html>.
- [5] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [7] MITRE. CWE CATEGORY: Concurrency. <https://cwe.mitre.org/data/definitions/557.html>.
- [8] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

- [10] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [11] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [12] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

