securing

www.securing.biz

Smart Contracts Security Review for Tellor Inc.

Report

document version: 1.0

author: Damian Rusinek (SecuRing)

test time period: 2021-09-07 – 2021-10-11

report date: 2021-10-13

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY

1.1. Summary of test results

- There are neither vulnerabilities with **critical** nor with **high** impact on risk found.
- There are **5** vulnerabilities with **medium** impact on risk found. Their potential consequences are:
  - Bypassing no-voting penalty and receiving the whole treasury reward (4.1).
  - Lagging voting system (4.1).
  - Lock of the funds used to buy treasury (4.2).
  - Paying different fee that specified in whitepaper (4.3).
  - Invalid business logic flow in the external application (4.4).
  - Initializing contract more than once with new parameters (4.5).
- There are **4** vulnerabilities with **low** impact on risk found. Their potential consequences are:
  - Denial of Service of the integrating contracts (4.6).
  - Temporal denial of service (4.7).
  - Losing control over the owner address and no possibility to initialize new version (4.8).
  - Use of an incorrect implementation (4.9).
- Additionally, **10 recommendations** have been proposed that do not have any direct risk impact. However, it is suggested to implement them due to good security practices.
- The upgrade process has been reviewed and no critical issues has been found.

1.2. Compliance with Smart Contract Security Verification Standard

Smart Contracts:	TellorX smart contracts	
Security Auditors:	Damian Rusinek, Pawel Kurylowicz	
Verification date:	08.10.2021r.	
Passed:	83	
Failed:	29	
Not applicable:	18	
Total score:	74%	83/112

The categories in which some of the requirements has not been met are:

- V1: Architecture, Design and Threat Modelling
- V2: Access Control
- V3: Blockchain Data

- V4: Communications
- V5: Arithmetic
- V6: Malicious Input Handling
- V8: Business Logic
- V9: Denial of Service
- V10: Token
- V11: Code Clarity
- V12: Test Coverage
- V13: Known Attacks
- V14: Decentralized Finance

None of the failed checks currently have a critical impact on the security of the system. The results of SCSVS compliance verification are attached to the report in the *SCSVS_v_12_Tellor.xlsx* file.

2. PROJECT OVERVIEW

2.1. Project description

The analyzed system is a new version of oracle system. It continues to work with crypto-economic incentives to retrieve off chain data, but introduce new uses of the systems monetary policy and the governance system. It distinguishes roles such as *reporter*, *voter*, *tipper*, *disputer*, *user*.

- ***reporter*** - submits new values to existing questions represented by ids,
- ***tipper*** – submits new question using its generated id and sends a tip that will be distributed among reporters,
- ***disputer*** – opens a dispute when an invalid value is submitted,
- ***voter*** – sends a vote for or against an open dispute or ongoing update proposal (decentralized governance member),
- ***user*** – retrieves values for particular question using its id.

An example usage scenario:

1. *User* (becoming a *tipper*) submits a question id and sends some TRB to incentive reporters.
2. *Reporters* submit many values assigned to the id.
3. *Disputer* does not agree on one value and opens a dispute.
4. *Voters* send votes to decide whether the value is correct or not.
5. *Disputer* wins the voting (one of the options) and the malicious *reporter* is slashed while *disputers* gets a fee.
6. *Users* retrieve the current value of the id.

2.2. Target in scope

The object being analysed were selected smart contracts accessible in the following:

- GitHub repository: https://github.com/tellor-io/tellorX/releases/tag/v0.1
- Commit ID: *cb3eafb54b4e6b728f6830dcd81e5b2b054e0095*

The contracts reviewed were the following:

- *Controller* – defines the functionality for changing contract addresses, as well as minting and migrating tokens.
- *Governance* – defines the functionality for proposing and executing votes, handling vote mechanism for voters, and distributing funds for initiators and disputed reporters depending on result.
- *Oracle* – defines the functionality for the *Tellor* oracle, where *reporters* submit values on chain and users can retrieve values.

- *TellorStaking* – defines the functionality for updating staking statuses for reporters, including depositing and withdrawing stakes.
- *TellorVars* – helper contract to store hashes of variables.
- *Token* – contains the methods related to transfers and ERC20, its storage and hashes of *Tellor* variables that are used to save gas on transactions.
- *Transition* – links to the *Oracle* contract and allows parties (like *Liquity*) to continue to use the master address to access values. All parties should be reading values through this address.
- *Treasury* – defines the function for *Tellor* treasuries.
- */tellor3*
    - *TellorMaster* – delegate calls to the *Tellor* contract. The logic implemented by this contract is saved on the *TellorGetters*, *TellorStake*, *TellorTransfer*, and *Extension* contracts.
    - *TellorStorage* – contains all the variables/structs used by *Tellor*
    - *TellorVariables* – helper contract to store hashes of variables

Additionally, the upgrade process was checked. During which the following things were verified:

- Appropriate modifiers for initializing functions to make sure they can only be called once and by authorized role.
- The order of state variables before and after the upgrade.
- Existence of reserved gap slots for future use and its correct update.
- Running and verifying the upgrade process on the local mainnet fork.
- Not front-runnable upgrade process.

2.3. Threat analysis

This section summarized the potential threats that were identified during initial threat modeling. The audit was mainly focused on, but not limited to, finding security issues that be exploited to achieve these threats.

The key threats were identified from particular perspectives as follows:

1. General (apply to all of the roles mentioned)
    - Bypassing the business logic of the system.
    - Theft of users' funds.
    - Errors in arithmetic operation.
    - Possibility to brick the contract.
    - Possibility to lock users' funds in the contract.
    - Incorrect access control for roles used by the contracts.
    - Existing of known vulnerabilities (e.g. front-running, re-entrancy, overflows)
2. Governance

- Too much power in relation to the declared one.
- Unintentional loss of the ability to governance the system.
- Immediate execution of submitted proposal.
- Gaining more voting power (called weights).

3. Treasury
   - Theft of funds from treasuries.
   - Bypassing the limits of treasuries and gaining higher profit.
4. Oracle
   - Submitting an invalid value that is accepted.
   - Performing a DoS attack on specific ids.
   - Gaining more or paying less fee due to calculation issues.
5. Staking
   - Bypassing the timelocks on withdrawals.
   - Submitting without staking.
6. Token
   - Base threats for ERC20 tokens.
   - Unauthorized mint and burn of tokens.
   - Bypassing the logic of limiting the transferable amount for stakers.

2.4. Testing overview

The security review of the *TellorX* smart contracts were meant to verify whether the proper security mechanisms were in place to prevent users from abusing the contracts' business logic and to detect the vulnerabilities which could cause financial losses to the client or its customers.

Security tests were performed using the following methods:

- Source code review,
- Automated tests through various tools (static analysis),
- Custom scripts (e.g. unit tests) for test scenarios based on key threats,
- Verification of compliance with Smart Contracts Security Verification Standard (SCSVS) in a form of code review,
- Q&A sessions with the client's representatives which allowed to gain knowledge about the project and the technical details behind the platform.

2.5. Basic information

Testing team	Damian Rusinek Paweł Kuryłowicz
Testing time period	2021-09-07 – 2021-10-11
Report date	2021-10-11 – 2021-10-13
Version	1.0

2.6. Disclaimer

The security review described in this report is not a security warranty.
It does not guarantee the security or correctness of reviewed smart contracts. Securing smart contract platforms is a multi-stage process, starting from threat modeling, through development based on best security practices, security reviews and formal verification, ending with constant monitoring and incident response.

Therefore, we strongly recommend implementing security mechanisms at all stages of development and maintenance.

3. SUMMARY OF IDENTIFIED VULNERABILITIES

3.1. Risk classification

Vulnerabilities	
Risk impact	**Description**
Critical	Vulnerabilities that affect the security of the entire system, all users or can lead to a significant financial loss (e.g. tokens). It is recommended to take immediate mitigating actions or limit the possibility of vulnerability exploitation.
High	Vulnerabilities that can lead to escalation of privileges, a denial of service or significant business logic abuse. It is recommended to take mitigating actions as soon as possible.
Medium	Vulnerabilities that affect the security of more than one user or the system, but might require specific privileges, or custom prerequisites. The mitigating actions should be taken after eliminating the vulnerabilities with critical and high risk impact.
Low	Vulnerabilities that affect the security of individual users or require strong custom prerequisites. The mitigating actions should be taken after eliminating the vulnerabilities with critical, high, and medium risk impact.
Recommendations	
Methods of increasing the security of the system by implementing good security practices or eliminating weaknesses. No direct risk impact has been identified. The decision whether to take mitigating actions should be made by the client.	

3.2. Identified vulnerabilities

Vulnerability	Risk impact	Status
4.1 No-voting penalty bypass	**Medium**	**Fixed**
4.2 Non-withdrawable treasury	**Medium**	**Fixed**
4.3 Non-compliant fees calculation	**Medium**	**Fixed**
4.4 Outdated data in events	**Medium**	
4.5 Lack of proper requirements for *init* function	**Medium**	
4.6 Not implemented functions	**Low**	**Partially fixed**
4.7 Unintended revert	**Low**	
4.8 Lack of new owner and deity validation	**Low**	
4.9 Invalid Main Tellor address hard-coded	**Low**	
Recommendations		
5.1 Consider additional safeguards against the known approve function problem		
5.2 Improve code clarity		
5.3 Consider optimizing indicated functions		
5.4 Fix calculations for small numbers		
5.5 Consider using constants for status values		
5.6 Consider following Check-Effects-Interactions		
5.7 Set variables visibility explicitly		
5.8 Validate parameters and set clear limits		
5.9 Inherit from interfaces for consistency		
5.10 Consider protecting killContract function		

4. VULNERABILITIES

4.1. No-voting penalty bypass

Risk impact	Medium	SCSVS V8
Vulnerable contract	*Treasury*	
Exploitation conditions	None	
Exploitation results	The investor who bought treasury and has not been voting since then can bypass no-voting penalty (Treasury.sol#L131-148) and receive the whole treasury reward. Additionally, another side consequence is that users might not be incentivized to take part in voting and voting system could be lagging.	

Remediation	Add a requirement in *buyTreasury* function to buy more than zero amount:
	`require(_amount > 0, "Amount must be greater than zero.");`
	Update the *startVoteCount* variable for the buyer only on the first buy:
	`if(_treas.accounts[msg.sender].amount == 0) {`
	`_treas.accounts[msg.sender].startVoteCount = IGovernance(governanceContract).getVoteCount();`
	`}`	

Status 2021-10-07:
Fixed
The code has been fixed during the tests.
Commit ID: *f7fa5f8fe355d31561fb8a884574ca694dc948c4*.

Vulnerability description:
The treasury contract allows users to buy issued treasuries and requires them to take part in the future voting. According to the documentation, the buyer's treasury profit is equal to the user's share in voting (the more they vote, the bigger share is, the higher profit is). The reduction calculation is present in *payTreasury* function in Treasury.sol#L131-148. The vulnerability allows the buyer to reset their *startVoteCount* variable right before paying back the treasury and get whole profit without the reduction.

Test case:
1. The user buys some treasury using *buyTreasury* function (Treasury.sol#L53) and protocol sets their *startVoteCount* variable in *accounts* mapping (Treasury.sol#L62).
2. The buyer waits until treasury is expired and do not take part in any voting.

3. Treasury is finished and the buyer is ready to pay back his amount (without profit because of not taking part in voting).
4. The buyer buys 0 amount using *buyTreasury* function (Treasury.sol#L53). Note that even if the treasury is sold out the buyer still can buy 0 amount.
    a. The protocol updates their *startVoteCount* variable in *accounts* mapping (Treasury.sol#L62).
5. The buyer calls *payTreasury* function (Treasury.sol#105) and gets the whole profit because their voting period is empty and contains no voting.

References:
SCSVS V8: Business logic
https://securing.github.io/SCSVS/1.2/0x17-V8-Business-Logic.html

4.2. Non-withdrawable treasury

Risk impact	Medium	SCSVS V8
Vulnerable contract	*Treasury*	
Exploitation conditions	Buying a treasury that has already expired.	
Exploitation results	Lock of the funds used to buy treasury.	
Remediation	Add a requirement in *buyTreasury* function to buy a treasury that is not yet expired: `require(_treas.dateStarted + _treas.duration > block.timestamp, "Treasury duration has expired.");`	

Status 2021-10-07:
Fixed
The code has been fixed during the tests.
Commit ID: *f7fa5f8fe355d31561fb8a884574ca694dc948c4*.

Vulnerability description:
The treasury contract allows users to buy treasury multiple times and at any time, including the case when user has already got back their treasury with profit. On the other hand, user can get back their treasury only once (Treasury.sol#L53). Therefore, when user buys a treasury that was already paid back, their funds are locked.

Test case:
1. The user buys some treasury using *buyTreasury* function (Treasury.sol#L53).
2. The buyer waits until treasury is expired.
3. The buyer calls *payTreasury* function (Treasury.sol#105) and gets the profit and protocol sets his treasury as paid back (Treasury.sol#L156).
4. The buyer buys more treasury using *buyTreasury* function (Treasury.sol#L53).

5. The buyer cannot withdraw their treasury anymore, because the requirement in Treasury.sol#L110 is not passed.

References:
SCSVS V8: Business logic
https://securing.github.io/SCSVS/1.2/0x17-V8-Business-Logic.html

4.3. Non-compliant fees calculation

Risk impact	Medium	SCSVS V5
Vulnerable contract	*Governance*	
Exploitation conditions	None	
Exploitation results	Paying different fee that specified in whitepaper.	
Remediation	Implement correct fee calculation: • Add minimum fee and set it to 10 TRB. • Correct the calculation of reducer and check whether the final fee is lower than minimum fee: `if(_stakeCount > 0){ _reducer = ((_stakeAmt - _minFee) * (_stakeCount * 1000)/_trgtMiners)/1000; } if(_reducer >= _stakeAmt - _minFee){ disputeFee = _minFee; }`	

Status 2021-10-07:
Fixed
The code has been fixed during the tests.
Commit ID: *53b35326587d2c241ef2a6eb9bf3a3deda1a993e.*

Vulnerability description:
The governance contract allows users to open disputes after sending a automatically calculated fee (depending on the stake amount and number of stakers). However, the fee update function (Governance.sol#L384) does not comply with the white paper in the following cases:

- Minimum fee (in WP) is 10 TRB, while in code it is 15 TRB (Governance.sol#L394).
- Contract calculates the reducer value (*_reducer*) in relation to the whole stake amount and checks whether the final fee is greater than zero (Governance.sol#L393) while it should check whether it is below minimum fee (10 TRB). This allows to make fee below the minimum (when the number of stakers is close to target stakers number).

Test case:
Below is the fragment of *updateMinDisputeFee* function (Governance.sol#L384):

```
(…)
if(_stakeCount > 0){
    _reducer = (_stakeAmt * (_min(_trgtMiners, _stakeCount) *
1000)/_trgtMiners)/1000;
}

if(_reducer >= _stakeAmt){
    disputeFee = 15e18;
}else{
    disputeFee = _stakeAmt - _reducer;
}
(…)
```

References:
SCSVS V5: Arithmetic
https://securing.github.io/SCSVS/1.2/0x14-V5-Arithmetic.md

4.4. Outdated data in events

Risk impact	Medium	SCSVS V4
Vulnerable contract	*Governance*	
Exploitation conditions	External application uses events as a source of information about voting that had multiple rounds.	
Exploitation results	Invalid business logic flow in the external application.	
Remediation	Emit the correct value in VoteExecuted event.	

Vulnerability description:
The *executeVote* function in Governance contract is used for executing passed proposal and disputes, however the voting result can be disputed itself and the next round of voting is started. After multiple rounds (minimum 2), when there is no other dispute open the vote is finally executed, fees are distributed among the reporters of each round and VoteExecuted is emitted.
The bug here is that in Governance.sol#L275 function emits an event with result (*_thisVote.result*) from the first round (instead of last round) because the value *_thisVote* is updated in a loop (Governance.sol#L268).

Test case:
Below is the fragment of *executeVote* function (Governance.sol#L268) with incorrect event data because the last value *_thisVote* variable is the first round of voting:

```
(…)
}else if(_thisVote.result == VoteResult.FAILED){
```

```
    // If vote is in dispute and fails, iterate through each vote round and
transfer the dispute to disputed reporter
    for(_i=voteRounds[_thisVote.identifierHash].length;_i>0;_i--){
        _voteID = voteRounds[_thisVote.identifierHash][_i-1];
        _thisVote = voteInfo[_voteID];
        _controller.transfer(_thisDispute.reportedMiner,_thisVote.fee);
    }
    _controller.changeStakingStatus(_thisDispute.reportedMiner,1);
}
emit VoteExecuted(_id, _thisVote.result);
```

References:
SCSVS V4: Communications
https://securing.github.io/SCSVS/1.2/0x13-V4-Communications.md

4.5. Lack of proper requirements for *init* function

Risk impact	Medium	SCSVS V4
Vulnerable contract	*Transition*	
Exploitation conditions	Being owner of the *Transition* contract.	
Exploitation results	Initializing contract more than once with new parameters.	
Remediation	Add a requirements for the *_governance* input parameter to be not equal to *address(0)*.	

Vulnerability description:
The *init* function (Transition.sol#L25) does not have sufficient requirements and allows to call itself more than once.

Test case:

- Transition.sol#L25-36

```
    function init(address _governance, address _oracle, address _treasury)
external{
        // Ensure sender is owner and transaction only occurs once
        require(msg.sender == addresses[_OWNER], "Only the owner address can
initiate a transition");
        require(addresses[_GOVERNANCE_CONTRACT] == address(0), "Only good once");
        // Set state amount and switch time
        uints[_STAKE_AMOUNT] = 100e18;
        uints[_SWITCH_TIME] = block.timestamp;
        // Define contract addresses
        addresses[_GOVERNANCE_CONTRACT] = _governance;
        addresses[_ORACLE_CONTRACT] = _oracle;
        addresses[_TREASURY_CONTRACT] = _treasury;
    }
```

References:
SCSVS V6: Malicious input handling

https://securing.github.io/SCSVS/1.2/0x15-V6-Malicious-Input-Handling.md

4.6. Not implemented functions

Risk impact	Low	SCSVS V1
Vulnerable contract	*Oracle*, *Controller*, *Treasury*, *Governance*	
Exploitation conditions	Third party contracts integrate with TellorX.	
Exploitation results	Denial of Service of the integrating contracts.	
Remediation	TellorX contracts should inherit from their interfaces: • IController, • IGovernance, • IOracle, • ITreasury. Remove *changeTypeInformation* function from allowed functions in Governance contract (Governance.sol#L83) and from IGovernance interface.	

Status 2021-10-07:
Partially fixed
The *changeTypeInformation* function has been removed during the tests.
Commit ID: *48d60555de8131579d8fa4eb001e228ffc9f2121*.

Vulnerability description:
TellorX contracts include interfaces that are used to cast addresses to particular contracts and call specific functions on them. However, the contract implementations do not inherit from the interfaces and there is a case where a contract does not implement all interface functions:

- Function *changeTypeInformation(uint256,uint256,uint256)* in IGovernance not implemented in Governance contract.

Note: The TellorX team decided that this function should not be implemented anymore. The recommendation is to remove it from the code.

Test case:
The list of functions allowed to be called by governance contract (Governance.sol#L75):

```
        bytes4[11] memory _funcs = [
            bytes4(0x3c46a185), // changeControllerContract(address)
            0xe8ce51d7, // changeGovernanceContract(address)
            0x1cbd3151, // changeOracleContract(address)
            0xbd87e0c9, // changeTreasuryContract(address)
            0x740358e6, // changeUint(bytes32,uint256)
            0x40c10f19, // mint(address,uint256)
            0xe48d4b3b, // setApprovedFunction(bytes4,bool)
            0xfad40294, // changeTypeInformation(uint256,uint256,uint256)
```

```
            0xe280e8e8, // changeMiningLock(uint256)
            0x6274885f, // issueTreasury(uint256,uint256,uint256)
            0xf3ff955a // delegateVotingPower(address)
        ];
```

The IGovernance interface (IGovernance.sol#L7):

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.3;

interface IGovernance{
    enum VoteResult {FAILED,PASSED,INVALID}
    function setApprovedFunction(bytes4 _func, bool _val) external;
    function changeTypeInformation(uint256 _id,uint256 _quorum, uint256
_duration) external;
    function beginDispute(bytes32 _id,uint256 _timestamp) external;
(…)
```

References:

SCSVS V1: Architecture, design and threat modeling
https://securing.github.io/SCSVS/1.2/0x10-V1-Architecture-Design-Threat-modelling.md

SCSVS V4: Communications
https://securing.github.io/SCSVS/1.2/0x13-V4-Communications.md

4.7. Unintended revert

Risk impact	Low	SCSVS V9
Vulnerable contract	*Governance*	
Exploitation conditions	Third party contracts integrate with TellorX and use *getDelegateInfo* function (Governance.sol#L450) or using this function internally.	
Exploitation results	Temporal denial of service.	
Remediation	Check whether there exist delegate info for holder.	

Vulnerability description:

Governance contract allows to get information on a delegate for a given holder using *getDelegateInfo* function (Governance.sol#L450). However, the function reverts if there was no delegation for the holder, because an underflow error happens.

Test case:

The *getDelegateInfo* function (Governance.sol#L450):

```
function getDelegateInfo(address _holder) external view returns(address,uint){
    return (delegateOfAt(_holder,block.number),
delegateInfo[_holder][delegateInfo[_holder].length-1].fromBlock);
}
```

References:
SCSVS V9: Denial of service
https://securing.github.io/SCSVS/1.2/0x18-V9-Denial-Of-Service.md

4.8. Lack of new owner and deity validation

Risk impact	Low	SCSVS V6
Vulnerable contract	*TellorMaster*	
Exploitation conditions	Setting wrong value for new owner or deity parameter.	
Exploitation results	Losing control over the owner address and no possibility to initialize new version.	
Remediation	General one: User two-step approach where the first call sets the pending owner and the second call must be sent by pending owner and updates the owner to its address. Specific for the current case: • Use the owner address to initialize new version and set it to zero-address to give away control to governance and deity (parachute contract). • Create a procedure of careful use of the multisig contract that becomes deity under specific circumstances defined in parachute contract.	

Vulnerability description:
The master contract contains owner and deity addresses and there exists *changeOwner* (TellorMaster.sol#L64) and *changeDeity* (TellorMaster.sol#L55) functions that instantly update these addresses. Passing an invalid address leads to losing control over the contracts.
It does not pose any risk to the deity address at the moment because it is already set to parachute contract used to react to specific unexpected events. However, in the future the parachute contract might change the deity address to multisig address and the owner address is used in the new version to initialize TellorX contracts. Therefore, these functions must be used carefully (see note below) and a procedure for their use should be prepared.
Note: The master contract is already deployed therefore it is impossible to change its logic without changing its address. That is why the recommendation is to use this function carefully in upgrade process, renounce ownership at the end and prepare procedure to use the multisig contract.
Additionally, there exists a _PENDING_OWNER storage slot that is used in two-step approach but is never used.

Test case:
The *changeOwner* function (TellorMaster.sol#L64):

```
function changeOwner(address _newOwner) external {
    require(msg.sender == addresses[_OWNER]);
    addresses[_OWNER] = _newOwner;
}
```

The *changeDeity* function (TellorMaster.sol#L55):

```
function changeDeity(address _newDeity) external {
    require(msg.sender == addresses[_DEITY]);
    addresses[_DEITY] = _newDeity;
}
```

References:
SCSVS V6: Malicious input handling
https://securing.github.io/SCSVS/1.2/0x15-V6-Malicious-Input-Handling.md

4.9. Invalid Main Tellor address hard-coded

Risk impact	Low	SCSVS V4
Vulnerable contract	*Transition*	
Exploitation conditions	Vulnerability in the old implementation or building new structures based on incorrect implementation.	
Exploitation results	Use of an incorrect implementation.	
Remediation	Introduce a process to check that hardcoded values are valid and up to date before every update. Change hardcoded address to the current implementation (for now: 0x2754da26f634e04b26c4decd27b3eb144cf40582).	

Vulnerability description:
Addresses that need to be hardcoded in a contract should be verified to make sure that their values are correct and that no errors have been made, such as typos, a copy-paste wrong value, or typing a value that is no longer valid.

Test case:
Address hardcoded in the *fallback* function points to old implementation of the *Main Tellor* contract:

- Transition.sol#L305:

```
fallback() external {
        address addr = 0xdDB59729045d2292eeb8Ff96c46B8db53B88Daa2; // Main Tellor
address (Harcode this in?) 0x2754da26f634e04b26c4decd27b3eb144cf40582
```

For now, current implementation can be found under following address:

```
0x2754da26f634e04b26c4decd27b3eb144cf40582
```

References:
SCSVS V4: Communications
https://securing.github.io/SCSVS/1.2/0x13-V4-Communications.html

5. RECOMMENDATIONS

5.1. Consider additional safeguards against the known approve function problem

Description:
The *Token* contract implements ERC20 standard which contains approve function. The function allows to set a given allowance of sender's tokens to a defined spender.
Front-running the transaction that changes existing allowance makes it possible for the malicious spender to transfer victim's tokens before the approve function call is added to block and again, using another transaction, to transfer the newly set amount of tokens.
As the consequence, the attacker has transferred the sum of both allowances and not the second allowance only, as it was desired by the victim.
The vulnerable implementation of *approve* function in *Token* contract can be found below.

- Token.sol#L66-71:

```
function approve(address _spender, uint256 _amount) external returns (bool) {
     require(_spender != address(0), "ERC20: approve to the zero address");
     _allowances[msg.sender][_spender] = _amount;
     emit Approval(msg.sender, _spender, _amount);
     return true;
}
```

This scenario presents a chain of transactions where the malicious spender was able to transfer 1100 tokens, while the maximum allowance being set by the victim was 1000:

```
Victim submits: approve(Spender, 1000) – ADDED TO MEMPOOL
Transaction validated: Victim: approve(Spender,1000)

#Victim wants to set allowance to 100 because 1000 was a mistake
Victim submits: approve(Spender, 100) – ADDED TO MEMPOOL

#Spender becomes MaliciousSpender
MaliciousSpender submits: transferFrom(Victim, 1000) – ADDED to MEMPOOL
Transaction validated: MaliciousSpender: transferFrom(Victim, 1000)
Transaction validated: Victim: approve(Spender, 100)
MaliciousSpender submits: transferFrom(Victim, 100) – ADDED TO MEMPOOL
Transaction validated: MaliciousSpender: transferFrom(Victim, 100)
```

How to implement:
As there is no general remediation for this vulnerability, we recommend to allow either set allowance:

```
to 0 (from any value)
```

or

```
from 0 (to any value).
```

References:
SCSVS V8: Business logic

https://securing.github.io/SCSVS/1.2/0x17-V8-Business-Logic.html
EIP-20
https://eips.ethereum.org/EIPS/eip-20

5.2. Improve code clarity

Description:
Due to the number and complexity of the components that make up the smart contract system, it is recommended to keep code clear and highly understandable. Describe your assumptions, reuse code when possible and not complicate the implementation.

Consider renaming the function from slashMiner to slashReporter
The function name does not match the new version role.

- TellorStaking.sol#L82:

```
function slashMiner(address _reporter, address _disputer) external{
(…)
```

Remove unnecessary code
In some places there is code that is never executed or is not needed for the proper functioning of the system.
Note: As of solidity 0.8.* over/underflows are automatically eliminated. Only the code in sections marked as **unchecked** might be susceptible to this type of vulnerability.

- Token.sol#202-205:

```
require(
        previousBalance + _sizedAmount >= previousBalance,
        "Overflow happened"
);
```

- Token.sol#221-230:

```
// Check for overflow for balance and supply
require(
        previousBalance + _sizedAmount >= previousBalance,
        "Overflow happened"
);
uint256 previousSupply = uints[_TOTAL_SUPPLY];
require(
        previousSupply + _amount >= previousSupply,
        "Overflow happened"
);
```

- Token.sol#251-260:

```
require(
       previousBalance - _sizedAmount <= previousBalance,
       "Overflow happened"
);
uint256 previousSupply = uints[_TOTAL_SUPPLY];
require(
        previousSupply - _amount <= previousSupply,
```

```
        "Overflow happened"
);
```

Some of the functions are not used in any way by the system or values might be retrieved from public variables.

- Governance.sol#L472-474

```
function getOpenDisputesOnId(bytes32 _id) external view returns(uint256){
        return openDisputesOnId[_id];
    }
```

- Governance.sol#L480-482

```
function getVoteCount() external view returns(uint256){
        return voteCount;
    }
```

- Governance.sol#L508-510

```
function getVoteRounds(bytes32 _hash) external view returns(uint256[] memory){
        return voteRounds[_hash];
    }
```

- Governance.sol#L517-519

```
function isFunctionApproved(bytes4 _func) external view returns(bool){
        return functionApproved[_func];
    }
```

- Oracle.sol#L172-174

```
function getMiningLock() external view returns(uint256){
        return miningLock;
    }
```

- Oracle.sol#L191-193

```
function getReportsSubmittedByAddress(address _reporter) external view
returns(uint256){
        return reportsSubmittedByAddress[_reporter];
    }
```

- Oracle.sol#L199-201

```
function getTimeBasedReward() external view returns(uint256){
        return timeBasedReward;
    }
```

- Oracle.sol#L226-228

```
function getTimeOfLastNewValue() external view returns(uint256){
        return timeOfLastNewValue;
    }
```

- Oracle.sol#L245-247

```
function getTipsById(bytes32 _id) external view returns(uint256){
        return tips[_id];
    }
```

- Oracle.sol#L254-256

```
function getTipsByUser(address _user) external view returns(uint256){
        return tipsByUser[_user];
    }
```

- Treasury.sol#L182-184

```
function getTreasuryCount() external view returns(uint256){
        return treasuryCount;
    }
```

- Treasury.sol#L195-197

```
function getTreasuryFundsByUser(address _user) external view returns(uint256){
        return treasuryFundsByUser[_user];
    }
```

Correct comments

Some of the contracts contain invalid comments, they do not match the actual action.

- Governance.sol#L97

```
    /**
     * @dev Helps initialize a dispute by assigning it a disputeId
     * when a miner returns a false/bad value on the validate array(in
Tellor.ProofOfWork) it sends the
     * invalidated value information to POS voting
     * @param _requestId being disputed
     * @param _timestamp being disputed
     */
```

- TellorStaking.sol#L64

```
    function requestStakingWithdraw() external {
        // Ensures reporter is not already staked
        StakeInfo storage stakes = stakerDetails[msg.sender];
        require(stakes.currentStatus == 1, "Reporter is not staked");
        // Change status to reflect withdraw request and updates start date for
staking
        stakes.currentStatus = 2;
        stakes.startDate = block.timestamp;
        // Update number of stakers and dispute fee
        uints[_STAKE_COUNT] -= 1;
        IGovernance(addresses[_GOVERNANCE_CONTRACT]).updateMinDisputeFee();
        emit StakeWithdrawRequested(msg.sender);
    }
```

- Transition.sol#L261

```
/**
* @dev Checks if a given hash of miner,requestId has been disputed
* @param _hash is the sha256(abi.encodePacked(_miners[2],_requestId,_timestamp));
* @return uint disputeId
*/
function getDisputeIdByDisputeHash(bytes32 _hash) external view returns (uint256)
{
     return disputeIdByDisputeHash[_hash];
}
```

Add events for important changes

Some functions that change the state, do not emit the appropriate events.

- Governance.sol#L175

```
function delegate(address _delegate) external{
        Delegation[] storage checkpoints = delegateInfo[msg.sender];
        // Check if sender hasn't delegated the specific address, or if the
current delegate is from old block number
        if (
            checkpoints.length == 0 ||
            checkpoints[checkpoints.length - 1].fromBlock != block.number
        ) {
            // Push a new delegate
            checkpoints.push(
                Delegation({
                    delegate: _delegate,
                    fromBlock: uint128(block.number)
                })
            );
        } else {
            // Else, update old delegate
            Delegation storage oldCheckPoint =
                checkpoints[checkpoints.length - 1];
            oldCheckPoint.delegate = _delegate;
        }
    }
```

Set the initial values explicitly

Not all values are initialized explicitly.

- Governance.sol#L388

```
function updateMinDisputeFee() public{
        uint256 _stakeAmt = IController(TELLOR_ADDRESS).uints(_STAKE_AMOUNT);
        uint256 _trgtMiners = IController(TELLOR_ADDRESS).uints(_TARGET_MINERS);
        uint256 _stakeCount = IController(TELLOR_ADDRESS).uints(_STAKE_COUNT);
        uint256 _reducer;
        // Calculate total dispute fee using stake count
(…)
```

- Treasury.sol#L112-113

```
    function payTreasury(address _investor,uint256 _id) external{
        // Validate ID of treasury, duration for treasury has not passed, and the
user has not paid
        TreasuryDetails storage treas = treasury[_id];
        require(_id <= treasuryCount, "ID does not correspond to a valid
treasury.");
        require(treas.dateStarted + treas.duration <= block.timestamp);
        require(!treas.accounts[_investor].paid);
        // Calculate non-voting penalty (treasury holders have to vote)
        uint256 numVotesParticipated;
        uint256 votesSinceTreasury;
        address governanceContract =
IController(TELLOR_ADDRESS).addresses(_GOVERNANCE_CONTRACT);
(…)
```

Fix variables names

The current name of the variable does not make it clear that this is the maximum amount.

- Treasury.sol#L32

```
    struct TreasuryDetails{
        uint256 dateStarted; // the date that treasury was started
        uint256 totalAmount; // the total amount stored in the treasury, in TRB
        uint256 rate; // the interest rate of the treasury, in BP
        uint256 purchased; // the amount of TRB purchased from the treasury
        uint256 duration; // the time in which the treasury locks participants
        uint256 endVoteCount; // the end vote count for when the treasury
duration is over
        bool endVoteCountRecorded; // determines if the vote count has been
calculated or not
        address[] owners; // the owners of the treasury
        mapping(address => TreasuryUser) accounts; // a mapping of a treasury
user address and corresponding details
    }
```

Add message to require

Some of the require statements do not contain the failure message.

- Governance.sol#L145

```
require(IController(TELLOR_ADDRESS).approveAndTransferFrom(msg.sender,address(thi
s), _fee));
```

- Treasury.sol#L55

```
require(IController(TELLOR_ADDRESS).approveAndTransferFrom(msg.sender,address(thi
s),_amount));
```

Set vote result explicitly

In one of the if statements, the voting result is not set explicitly.

- Governance.sol#L370

```
else if (_thisVote.doesSupport > _thisVote.against) {
     if (_thisVote.doesSupport >=
((IController(TELLOR_ADDRESS).uints(_TOTAL_SUPPLY) * _quorum) / 100)) {
                _thisVote.result = VoteResult.PASSED;
                Dispute storage _thisDispute = disputeInfo[_id];
                (uint256 _status,) =
IController(TELLOR_ADDRESS).getStakerInfo(_thisDispute.reportedMiner);
                if(_thisVote.isDispute && _status == 3){
IController(TELLOR_ADDRESS).changeStakingStatus(_thisDispute.reportedMiner,4);
                }
            } [lack of explicit set for vote result]
        }
```

Push msg.sender to _treas.owners only if it does not exits

There is no verification if the *msg.sender* address already exist in the *_treas.owners*

- Treasury.sol#L53-68

```
    function buyTreasury(uint256 _id,uint256 _amount) external {
        // Transfer sender funds to Treasury
```

```
require(IController(TELLOR_ADDRESS).approveAndTransferFrom(msg.sender,address(thi
s),_amount));
        treasuryFundsByUser[msg.sender]+=_amount;
        // Check for sufficient treasury funds
        TreasuryDetails storage _treas = treasury[_id];
        require(_amount <= _treas.totalAmount - _treas.purchased, "Not enough
money in treasury left to purchase.");
        // Update treasury details -- vote count, purchased, amount, and owners
        address governanceContract =
IController(TELLOR_ADDRESS).addresses(_GOVERNANCE_CONTRACT);
        _treas.accounts[msg.sender].startVoteCount =
IGovernance(governanceContract).getVoteCount();
        _treas.purchased += _amount;
        _treas.accounts[msg.sender].amount += _amount;
        _treas.owners.push(msg.sender);
        totalLocked += _amount;
        emit TreasuryPurchased(msg.sender,_amount);
    }
```

How to implement:

Consider renaming the function from slashMiner to slashReporter

Rename function to *slashReporter*.

Note: Be aware that this may affect compatibility with the previous versions or interfaces

```
function slashReporter(address _reporter, address _disputer) external{
[…]
```

Remove unnecessary code

We encourage to remove any unnecessary code.

Correct comments

Some of the contracts contain invalid comments, they do not match the actual action.

- Governance.sol#L97

```
    /**
     * @dev Helps initialize a dispute by assigning it a disputeId
     * @param _requestId being disputed
     * @param _timestamp being disputed
     */
```

- TellorStaking.sol#L64

```
    function requestStakingWithdraw() external {
        // Ensures reporter is staked
        StakeInfo storage stakes = stakerDetails[msg.sender];
        require(stakes.currentStatus == 1, "Reporter is not staked");
        // Change status to reflect withdraw request and updates start date for
staking
        stakes.currentStatus = 2;
        stakes.startDate = block.timestamp;
        // Update number of stakers and dispute fee
        uints[_STAKE_COUNT] -= 1;
        IGovernance(addresses[_GOVERNANCE_CONTRACT]).updateMinDisputeFee();
        emit StakeWithdrawRequested(msg.sender);
```

```
    }
```

- Transition.sol#L261

```
/**
* @dev Gets id if a given hash of miner has been disputed.
* @param _hash is the sha256(abi.encodePacked(_miners[2],_requestId,_timestamp));
* @return uint disputeId
*/
function getDisputeIdByDisputeHash(bytes32 _hash) external view returns (uint256)
{
     return disputeIdByDisputeHash[_hash];
}
```

Add events for important changes

Emit proper events on state changes.

```
function delegate(address _delegate) external{
        Delegation[] storage checkpoints = delegateInfo[msg.sender];
        // Check if sender hasn't delegated the specific address, or if the
current delegate is from old block number
        if (
            checkpoints.length == 0 ||
            checkpoints[checkpoints.length - 1].fromBlock != block.number
        ) {
            // Push a new delegate
            checkpoints.push(
                Delegation({
                    delegate: _delegate,
                    fromBlock: uint128(block.number)
                })
            );
        } else {
            // Else, update old delegate
            Delegation storage oldCheckPoint =
                checkpoints[checkpoints.length - 1];
            oldCheckPoint.delegate = _delegate;
        }
emit YOUR_EVENT_HERE
    }
```

Set the initial values explicitly

- Governance.sol#L388

```
function updateMinDisputeFee() public{
        uint256 _stakeAmt = IController(TELLOR_ADDRESS).uints(_STAKE_AMOUNT);
        uint256 _trgtMiners = IController(TELLOR_ADDRESS).uints(_TARGET_MINERS);
        uint256 _stakeCount = IController(TELLOR_ADDRESS).uints(_STAKE_COUNT);
        uint256 _reducer = 0;
        // Calculate total dispute fee using stake count
(…)
```

- Treasury.sol#L112-113

```
    function payTreasury(address _investor,uint256 _id) external{
        // Validate ID of treasury, duration for treasury has not passed, and the
user has not paid
        TreasuryDetails storage treas = treasury[_id];
```

```
        require(_id <= treasuryCount, "ID does not correspond to a valid
treasury.");
        require(treas.dateStarted + treas.duration <= block.timestamp);
        require(!treas.accounts[_investor].paid);
        // Calculate non-voting penalty (treasury holders have to vote)
        uint256 numVotesParticipated = 0;
        uint256 votesSinceTreasury = 0;
        address governanceContract =
IController(TELLOR_ADDRESS).addresses(_GOVERNANCE_CONTRACT);
(…)
```

Fix variables names

Change *TreasuryDetails.*totalAmount to *TreasuryDetails.*maxAmount

Add message to require

Add failure message to indicated require statements.

Set vote result explicitly

- Governance.sol#L370

```
else if (_thisVote.doesSupport > _thisVote.against) {
    if (_thisVote.doesSupport >=
((IController(TELLOR_ADDRESS).uints(_TOTAL_SUPPLY) * _quorum) / 100)) {
                _thisVote.result = VoteResult.PASSED;
                Dispute storage _thisDispute = disputeInfo[_id];
                (uint256 _status,) =
IController(TELLOR_ADDRESS).getStakerInfo(_thisDispute.reportedMiner);
                if(_thisVote.isDispute && _status == 3){
IController(TELLOR_ADDRESS).changeStakingStatus(_thisDispute.reportedMiner,4);
                }
    } else {
        _thisVote.result = VoteResult.FAILED;
    }
}
```

Push msg.sender to _treas.owners only if it does not exits

- Treasury.sol#L53-68

```
    function buyTreasury(uint256 _id,uint256 _amount) external {
        // Transfer sender funds to Treasury

require(IController(TELLOR_ADDRESS).approveAndTransferFrom(msg.sender,address(thi
s),_amount));
        treasuryFundsByUser[msg.sender]+=_amount;
        // Check for sufficient treasury funds
        TreasuryDetails storage _treas = treasury[_id];
        require(_amount <= _treas.totalAmount - _treas.purchased, "Not enough
money in treasury left to purchase.");
        // Update treasury details -- vote count, purchased, amount, and owners
        address governanceContract =
IController(TELLOR_ADDRESS).addresses(_GOVERNANCE_CONTRACT);
        _treas.accounts[msg.sender].startVoteCount =
IGovernance(governanceContract).getVoteCount();
        _treas.purchased += _amount;
        _treas.accounts[msg.sender].amount += _amount;
        if(_treas.accounts[msg.sender].amount == 0) {
```

```
            _treas.owners.push(msg.sender);
        }
        totalLocked += _amount;
        emit TreasuryPurchased(msg.sender,_amount);
    }
```

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

5.3. Consider optimizing indicated functions

Description:
During the tests, 2 functions were detected that may be more gas efficient while maintaining their current operation.

- Transition.sol#L337-344

```
function _sliceUint(bytes memory b) internal pure returns (uint256 _x){
     uint256 number;
     for(uint256 i=0;i<b.length;i++){
         number = number + uint256(uint8(b[i]))*(2**(8*(b.length-(i+1))));
     }
     return number;
}
```

- Governance.sol#L267-273

```
(…)
// If vote is in dispute and fails, iterate through each vote round and transfer
the dispute to disputed reporter
for(_i=voteRounds[_thisVote.identifierHash].length;_i>0;_i--){
    _voteID = voteRounds[_thisVote.identifierHash][_i-1];
    _thisVote = voteInfo[_voteID];
    _controller.transfer(_thisDispute.reportedMiner,_thisVote.fee);
}
_controller.changeStakingStatus(_thisDispute.reportedMiner,1);
(…)
```

How to implement:
Consider one of the indicated implementations. Before applying it, make sure that it works as intended.

- The *_sliceUint* function

```
function sliceUintOptimized(bytes memory b) public pure returns (uint256 _x){
        uint256 number = 0;
        for(uint256 i=0;i<b.length;i++){
            number = number * 2**8;
            number = number + uint8(b[i]);
        }
        return number;
}
```

- The *executeVote* function

```
(…)
// If vote is in dispute and fails, iterate through each vote round and transfer
the dispute to disputed reporter
uint256 reporterReward = 0;
for(_i=voteRounds[_thisVote.identifierHash].length;_i>0;_i--){
    _voteID = voteRounds[_thisVote.identifierHash][_i-1];
    _thisVote = voteInfo[_voteID];
    reporterReward += _thisVote.fee;
}
_controller.transfer(_thisDispute.reportedMiner, reporterReward);
_controller.changeStakingStatus(_thisDispute.reportedMiner,1);
(…)
```

References:

V7: Gas usage & limitations

https://securing.github.io/SCSVS/1.2/0x16-V7-Gas-Usage-And-Limitations.md

5.4. Fix calculations for small numbers

Description:

Some of the calculations in the contract do not maintain the proper precision of calculations and may lead to incorrect operation.

- Governance.sol#L144,147:

```
// Calculate dispute fee based on number of current vote rounds
uint256 _fee;
if(voteRounds[_hash].length == 1){
    _fee = disputeFee * 2**(openDisputesOnId[_requestId] - 1);
    IOracle(_oracle).removeValue(_requestId,_timestamp);
}
else{
    _fee = disputeFee * 2**(voteRounds[_hash].length - 1);
}
_thisVote.fee = _fee * 9 / 10;
require(IController(TELLOR_ADDRESS).approveAndTransferFrom(msg.sender,
address(this), _fee)); // This is the fork fee (just 100 tokens flat, no refunds.
Goes up quickly to dispute a bad vote)
// Add an initial tip and change the current staking status of reporter
IOracle(_oracle).addTip(_requestId,_fee/10);
IController(TELLOR_ADDRESS).changeStakingStatus(_reporter,3);
emit NewDispute(_id, _requestId, _timestamp, _reporter);
```

How to implement:

To keep the accuracy of the calculations, we propose to subtract fee values:

```
// Calculate dispute fee based on number of current vote rounds
uint256 _fee;
if(voteRounds[_hash].length == 1){
    _fee = disputeFee * 2**(openDisputesOnId[_requestId] - 1);
    IOracle(_oracle).removeValue(_requestId,_timestamp);
}
else{
    _fee = disputeFee * 2**(voteRounds[_hash].length - 1);
}
```

```
_thisVote.fee = _fee * 9 / 10;
require(IController(TELLOR_ADDRESS).approveAndTransferFrom(msg.sender,
address(this), _fee)); // This is the fork fee (just 100 tokens flat, no refunds.
Goes up quickly to dispute a bad vote)
// Add an initial tip and change the current staking status of reporter
IOracle(_oracle).addTip(_requestId,_fee -_thisVote.fee);
IController(TELLOR_ADDRESS).changeStakingStatus(_reporter,3);
emit NewDispute(_id, _requestId, _timestamp, _reporter);
```

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html
Math in Solidity:
https://medium.com/coinmonks/math-in-solidity-part-1-numbers-384c8377f26d

5.5. Consider using constants for status values

Description:
Currently, status is implemented as uint256 in the *StakeInfo* struct. The individual numbers correspond to the appropriate status, what is not easy to read.

- TellorStorage.sol#L27-30:

```
struct StakeInfo {
    uint256 currentStatus; //0-not Staked, 1=Staked, 2=LockedForWithdraw 3=
OnDispute 4=ReadyForUnlocking 5=Unlocked
    uint256 startDate; //stake start date
}
```

How to implement:
Consider using an *enum* with constant status variants that are easy to understand and reduce the risk of confusion:

```
enum currentStatus{
    NotStaked,
    Staked,
    LockedForWithdraw,
    OnDispute,
    ReadyForUnlocking,
    Unlocked
}
```

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

5.6. Consider following Check-Effects-Interactions

Description:
Currently external calls are made only to the trusted token contract which does not allow for reentrant behaviour. Future changes (e.g. supporting token transfer confirmation by receiver) might introduce unsafe external calls and vulnerabilities may appear.

How to implement:
Build a function logic based on the following scheme:

- First, check any requirements.
- Then, update state.
- Finally, perform interactions with external contracts.

References:
Reentrancy attack in smart contracts – is it still a problem?
https://www.securing.pl/en/reentrancy-attack-in-smart-contracts-is-it-still-a-problem/
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

5.7. Set variables visibility explicitly

Description:
The visibility of the identified variables has not been explicitly set. By default, they are set to public, but it is good practice to declare visibility directly.

- Oracle.sol#L22-24:

```
mapping(bytes32 => Report) reports;
mapping(address => uint256) reporterLastTimestamp;
mapping(address => uint256) reportsSubmittedByAddress;
mapping(address => uint256) tipsByUser;
```

- Governance.sol#L22-27:

```
mapping (address => Delegation[]) delegateInfo;
mapping(bytes4 => bool) functionApproved;
mapping(bytes32 => uint[]) voteRounds;
mapping(uint => Vote) voteInfo;
mapping(uint => Dispute) disputeInfo;
mapping(bytes32 => uint) openDisputesOnId;
```

How to implement:
Set visibility explicitly:

- Oracle.sol#L22-24:

```
mapping(bytes32 => Report) public reports;
mapping(address => uint256) public reporterLastTimestamp;
mapping(address => uint256) public reportsSubmittedByAddress;
mapping(address => uint256) public tipsByUser;
```

- Governance.sol#L22-27:

```
mapping (address => Delegation[]) public delegateInfo;
mapping(bytes4 => bool) public functionApproved;
mapping(bytes32 => uint[]) public voteRounds;
mapping(uint => Vote) public voteInfo;
mapping(uint => Dispute) public disputeInfo;
mapping(bytes32 => uint) public openDisputesOnId;
```

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

5.8. Validate parameters and set clear limits

Description:
Some of the parameters are not validated. It is good practise to enforce limits for expected values, even when those are set by governance.

- Oracle.sol#L66-70:

```
function changeMiningLock(uint256 _newMiningLock) external{
        require(msg.sender ==
IController(TELLOR_ADDRESS).addresses(_GOVERNANCE_CONTRACT), "Only governance
contract can change mining lock.");
        miningLock = _newMiningLock;
        emit MiningLockChanged(_newMiningLock);
}
```

- Oracle.sol#L77-81:

```
function changeTimeBasedReward(uint256 _newTimeBasedReward) external{
        require(msg.sender ==
IController(TELLOR_ADDRESS).addresses(_GOVERNANCE_CONTRACT), "Only governance
contract can change time based reward.");
        timeBasedReward = _newTimeBasedReward;
        emit TimeBasedRewardsChanged(_newTimeBasedReward);
}
```

- Treasury.sol#L87-97:

```
function issueTreasury(uint256 _totalAmount, uint256 _rate, uint256 _duration)
external{
        require(msg.sender ==
IController(TELLOR_ADDRESS).addresses(_GOVERNANCE_CONTRACT), "Only governance
contract is allowed to issue a treasury.");
        // Increment treasury count, and define new treasury and its details
(start date, total amount, rate, etc.)
        treasuryCount++;
        TreasuryDetails storage _treas = treasury[treasuryCount];
        _treas.dateStarted = block.timestamp;
        _treas.totalAmount = _totalAmount;
        _treas.rate = _rate;
        _treas.duration = _duration;
        emit TreasuryIssued(treasuryCount,_totalAmount,_rate);
    }
```

How to implement:
Set limits and enforce input parameter values to match their requirements.

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

5.9. Inherit from interfaces for consistency

Description:
Some of the contracts, although they are based on interface functionalities, do not inherit from it.

- Controller.sol#L15:

```
contract Controller is TellorStaking, Transition{
(…)
```

- Oracle.sol#L14:

```
contract Oracle is TellorVars{
(…)
```

- Treasury.sol#L15:

```
contract Treasury is TellorVars{
(…)
```

- Governance.sol#L18:

```
contract Governance is TellorVars{
(…)
```

How to implement:
The Treasury should inherit from ITreasury.
The Governance should inherit from IGovernance.
The Controller should inherit from IController.
The Oracle should inherit from IOracle.

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.2/0x20-V11-Code-Clarity.html

5.10. Consider protecting killContract function

Description:
There exists a parachute contract, already deployed on mainnet at address *0x83eB2094072f6eD9F57d3F19f54820ee0BaE6084*. This contract is out of scope however it is the current deity of the *TellorMaster* contract and have a big impact on the *Tellor* protocol.

The parachute is used to react to specific unexpected events (such as no reporting new oracle values for a long time) and bring back the control over the *Tellor* system to multisig address (using *changeDeity* function).
However, there exists a *killContract* function that changes the deity to zero-address and blocks all other function calls (reactions to unexpected events) because the master contract will not accept any sender but zero-address since then.

How to implement:
Prepare a procedure of careful use (for parachute contract) that prevents calling *killContract* function by mistake.

References:
SCSVS V2: Access control
https://securing.github.io/SCSVS/1.2/0x11-V2-Access-Control.md

6. SCSVS

The application has been verified for compliance with the Smart Contracts Security Verification Standard (SCSVS), v. 1.2.

Legend:

- Passed – The application meets the requirement.
- Failed – The application does not meet the requirements.
- N/A – The requirement does not apply to system (e.g. the system does not include the feature to which the requirement applies) or is out of scope.

The results of SCSVS compliance verification are attached to the report in the *SCSVS_v12_Tellor.xlsx* file.

6.1. Failed checks

The list of failed checks:

- V1: Architecture, design and threat modelling
  - Verify that the events for the (state changing/crucial for business) operations are defined
  - Verify that there is a component that monitors the contract activity using events.
  - Verify that if the fallback function can be called by anyone it is included in the threat modelling.
  - Verify that the latest version of the major Solidity release is used.
  - Verify that, when using the external implementation of contract, you use the current version which has not been superseded.
  - Verify that there are no vulnerabilities associated with system architecture and design.
- V2: Access Control
  - Verify that the initialization functions are marked internal and cannot be executed twice.
  - Verify that there are no vulnerabilities associated with access control.
- V3: Blockchain Data
  - Verify that there is a component that monitors access to sensitive contract data using events.
- V4: Communications
  - Verify that there are no vulnerabilities associated with communications.
- V5: Arithmetic
  - Verify that the extreme values (e.g. maximum and minimum values of the variable type) are considered and does change the logic flow of the contract.

    - Verify that there are no vulnerabilities associated with arithmetics.
- V6: Malicious input Handling
    - Verify that if the input (function parameters) is validated, the positive validation approach (allowlisting) is used where possible.
    - Verify that there are no vulnerabilities associated with malicious input handling.
- V8: Business Logic
    - Verify that the contract logic implementation corresponds to the documentation.
    - Verify that the business logic flows of smart contracts proceed in a sequential step order and it is not possible to skip any part of it or to do it in a different order than designed.
    - Verify that there are no vulnerabilities associated with business logic.
- V9: Denial of Service
    - Verify that the expressions of functions assert or require to have a passing variant.
    - Verify that there are no vulnerabilities associated with availability.
- V10: Token
    - Use the approve function from the ERC-20 standard to change allowed amount only to 0 or from 0.
- V11: Code Clarity
    - Verify that the contract uses existing and tested code (e.g. token contract or mechanisms like ownable) instead of implementing its own.
    - Verify that the same rules for variable naming are followed throughout all the contracts (e.g. use the same variable name for the same object).
    - Verify that all storage variables are initialised.
    - Verify that all functions are used. Unused ones should be removed.
- V12: Test Coverage
    - Verify that the specification of smart contract has been formally verified.
    - Verify that the specification and the result of formal verification is included in the documentation.
- V13: Known attacks
    - Verify that the contract is not vulnerable to Access Control issues.
    - Verify that the contract is not vulnerable to Denial of Service attacks.
- V14: Decentralized Finance
    - Verify that the smart contract attributes that can be updated by the external contracts (even trusted) are monitored (e.g. using events) and the procedure of incident response is implemented (e.g. the response to an ongoing attack).

7. TERMINOLOGY

This section explains the terms that are related to the methodology used in this report.

Threat

Any circumstance or event with the potential to adversely impact organizational operations (including mission, functions, image, or reputation), organizational assets, or individuals through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of service.[1]

Vulnerability

Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source.[1]

Risk

The level of impact on organizational operations (including mission, functions, image, or reputation), organizational assets, or individuals resulting from the operation of an information system given the potential impact of a threat and the likelihood of that threat occurring.[1]

The risk impact can be estimated based on the complexity of exploitation conditions (representing the likelihood) and the severity of exploitation results.

		Complexity of exploitation conditions		
		Simple	Moderate	Complex
Severity of exploitation results	Major	Critical	High	Medium
	Moderate	High	Medium	Low
	Minor	Medium	Low	Low

[1] NIST FIPS PUB 200: Minimum Security Requirements for Federal Information and Information Systems. Gaithersburg, MD: Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology.

8. CONTACT

Person responsible for providing explanations:

Damian Rusinek
e-mail: Damian.Rusinek@securing.pl
tel.: +48 12 425 25 75

securing
http://www.securing.pl
e-mail: info@securing.pl
Kalwaryjska 65/6
30-504 Kraków
tel./fax.: +48 (12) 425 25