Auditing report – David Bloomer

# 1. Summary

In this report I will be auditing the smart contract of Tomas Rokos. The goal is to look for and expose vulnerabilities in the code that could lead to potential dangers. As the implemented project tries to implement a secure blockchain voting system, the main vulnerability to look for is whether an outside party could influence the election results. The audit will be done based on the tools, techniques and methodology provided by the respected auditing firm Ackee, and by the course material of NIE-BLO.

Tools

* Woke

# 2. Methodology

1. System overview: brief explanation of the system, its functionalities and its actors.
2. Local deployment: check wether all the UC’s work by deploying the smart contract locally and testing the functions
3. Tool-based analysis with Woke
4. Manual code review: code is manually checked line by line to check for vulnerabilities

# 3. System overview

## 3.1 Contracts

IVoteD21.sol

This contract served as an interface for the other contract that is described below. It had a total of 8 functions that each needed to be implemented to provide full functionality for the voting system.

D21.sol

This contract implemented the 8 functions from IVoteD21.sol by using inheritance. The functions had to accomplish the following functionalities:

* Add a new subject into the voting system using the name.
  + function addSubject(string memory name) external;
* Add a new voter into the voting system.
  + function addVoter(address addr) external;
* Get addresses of all registered subjects.
  + function getSubjects() external view returns(address[] memory);
* Get the subject details.
  + function getSubject(address addr) external view returns(Subject memory);
* Vote positive for the subject.
  + function votePositive(address addr) external;
* Vote negative for the subject.
  + function voteNegative(address addr) external;
* Get the remaining time to the voting end in seconds.
  + function getRemainingTime() external view returns(uint);
* Get the voting results, sorted descending by votes.
  + function getResults() external view returns(Subject[] memory);

Alltogether, the implementation of the above functions had to lead to a full fledged voting system.

## 3.2 Actors

Owner

The owner is the of the contract is the address that deploys the contract. The owner is set by the constructor which is called when the contract is deployed. The owner is the only actor who can add eligible voters to the system. In reality, this could be the government or some sort of other trusted party. It has to be noted that, except from adding voters, the owner shouldn’t be able to influence the voting results.

Voters

The voters are the addresses that, after being allowed by the owner, can cast their votes on particular subjects. They have 2 positive votes, and after they used those 2, they can cast 1 negative vote. Once they have been added as an eligble voter, they can’t be removed.

Subject owners

Each address, no matter if they have been added to the eligible voters list or not, can add a subject on which voters can vote. Each address can only add one subject. Subjects can’t be removed.

## 3.3 Trusted party

For this smart contract, the owner is the trusted party because only he or she has the ability to add voters. It is therefore important that the owner isn’t partisan in his selection of voters.

# 4. Audit

## Local deployment

For the local deployment we manually checked if all the inteded UC’s were implemented. In the following, we will report whether it passed, and which function was responsible

* UC1 - Everyone can register a subject (e.g. political party)
  + Passed
  + addSubject()
* UC2 - Everyone can list registered subjects
  + Passed
  + getSubjects()
* UC3 - Everyone can see the subject’s results
  + Passed
  + getSubject() / getResults()
* UC4 - Only the owner can add eligible voters
  + Passed
  + addVoter()
* UC5 - Every voter has 2 positive and 1 negative vote
  + Passed
  + votePositive() / voteNegative()
* UC6 - Voter can not give more than 1 vote to the same subject
  + Passed
  + votePositive() / voteNegative()
* UC7 - Negative vote can be used only after 2 positive votes
  + Passed
  + voteNegative()
* UC8 - Voting ends after 7 days from the contract deployment
  + Passed
  + <whole system>

## Tooling output

Woke

By using the command *woke detect* the tool checked the smart contract for vulnerabilities. It used the following detectors:

Using the following detectors:

* axelar-proxy-contract-id
  + Detects incorrect use of the contractId function in Axelar proxy and upgradeable contracts.
* function-call-options-not-called
  + Function with gas or value set actually is not called, e.g. this.externalFunction.value(targetValue) or this.externalFunction{value: targetValue}.
* overflow-calldata-tuple-reencoding-bug
  + Detects Head Overflow Calldata Tuple Reencoding compiler bug
* reentrancy
  + Detects re-entrancy vulnerabilities.
* unchecked-function-return-value
  + Return value of a function call is ignored.
* unsafe-address-balance-use
  + Address.balance is either written to a state variable or used in a strict comparison (== or !=).
* unsafe-delegatecall
  + Delegatecall to an untrusted contract.
* unsafe-selfdestruct
  + Selfdestruct call is not protected.

The tool didn’t detect any vulnerabilities.

## C1: Anybody can add a subject with the same

|  |  |  |  |
| --- | --- | --- | --- |
| Impact: | Medium | Likelihood: | High |
| Target: | addSubject() | Type | incorrect business logic |

Description

The protocol allows different addresses to add subjects with the same name. There is no check to see if the subject name allready exists.

Exploit scenario

If one address (or person) added an original subject, let’s say ‘Party of Freedom’, there is nothing that holds other addresses back from adding a subject with the same name. This way, certain subjects can be boycotted by others because they can keep adding multiple subjects with the same name. The result is that potential voters could be confused about which subject is the original one, and therefore could vote on the non-original ones which would take votes away from the original idea.

The scenario is demonstrated below.

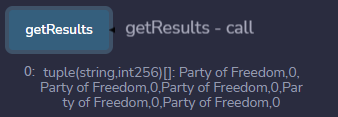


Figure 1: Multiple subjects with the same name

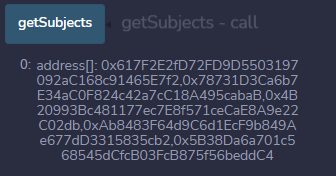
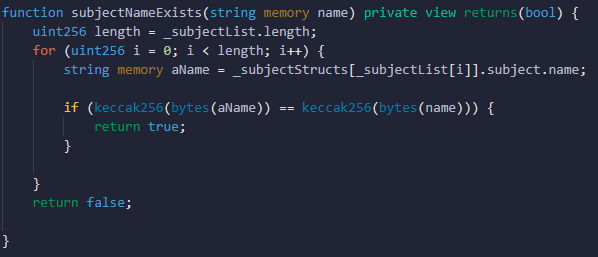


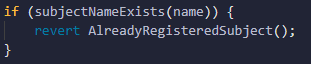
Figure 2: Multiple addresses for subjects with the same name

Recommendation

The simple solution to this problem is to reject new subjects that have a name that is allready registered. To achieve this, one could write the following code:



This function can than be called when adding a subject:



## C2: There is no limit on the length of the subject string that you can add

|  |  |  |  |
| --- | --- | --- | --- |
| Impact: | High | Likelihood: | High |
| Target: | addSubject() | Type | DoS |

Description

One of the functions is the addSubject() function which allows everyone to add one and only one subject, which consists of a string. The reason behind this is that in a democracy everyone should be able to add subjects. The problem is that there is no limit to the amount of characters the subject can have. You can therefore add a subjects that contains a high enough number of characters that it can lead to a crash.

Exploit scenario

Call the function addSubject(aString) whereby aString contains a very high amount of characters making it hard to process and cause problems.

Recommendation

Add a simple clause in the function addSubject() that requires the string of the subject to only be a certain length. For example:

function addSubject(string memory name) external onlyInVotingPeriod {

        require(bytes(name).length < 100,"Subject name is to long");

        SubjectEntity storage subject = \_subjectStructs[msg.sender];

        if (subject.exists) {

            revert AlreadyRegisteredSubject();

        }

## C3: The use of block.timestamp can be manipulated

|  |  |  |  |
| --- | --- | --- | --- |
| Impact: | Low | Likelihood: | Medium |
| Target: | onlyInVotingPeriod() | Type: |  |

Description

Dangerous usage of block.timestamp because it can be manipulated by miners which will cause an effect on the workings of the smart contract.

Exploit scenario

In the closing moments of the voting, the execution of certain functions heavily rely on block.timestamp (because of the condition onlyInVotingPeriod()). Miners can manipulate this and can thus manipulate the outcome of the voting.

Recommendation

Avoid relying on block.timestamp.

# 5. Conclusion

The goal of the smart contract was to implement a secure decentralized voting system. The most critical part of such a system therefore was to ensure that outside parties couldn’t influence the outcome of the voting. The audit that had to be performed thus mainly focussed on detecting possibilities of voter fraud. From a technical perspective the smart contract was relatively secure. Albeit two (C2, C3), the audit didn’t find any very prominent technical vulnerabilities that could lead to voter fraud. However, from a business logic perspective, the vulnerability C1 did have the ability to lead to major confusion which could drastically influence election results.