HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: StrongDeFi

Date: September 18nd, 2020

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

This document may contain confidential information about IT systems and the intellectual property of the customer as well as information about potential vulnerabilities and methods of their exploitation.

The report containing confidential information can be used internally by the customer or it can be disclosed publicly after all vulnerabilities fixed - upon a decision of the customer.

Document

Name	Smart Contract Code Review and Security Analysis Report for StrongDeFi
Type	DeFi Contracts
Platform	Ethereum / Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Archive Name	strongdefi-babramson-09-11-20.zip
SHA256 hash	d2f4ef45cf962b6601aa05009c2647c925d24ecce1bf2217789e2685846ab1c3
Timeline	11 SEP 2020 – 16 SEP 2020
Changelog	16 SEP 2020 - INITIAL AUDIT 18 SEP 2020 – REMEDIATIONS

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table of contents

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Introduction

Hacken OÜ (Consultant) was contracted by StrongDeFi (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of Customer's smart contract and its code review conducted between September 11, 2020 – September 16, 2020.

Scope

The scope of the project is smart contracts in the repository:
Audit Archive File – strongdefi-babramson-09-11-20.zip
SHA256 hash – d2f4ef45cf962b6601aa05009c2647c925d24ecce1bf2217789e2685846ab1c3
We have scanned this smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that are considered:

Category	Check Item
Code review	▪ Reentrancy ▪ Ownership Takeover ▪ Timestamp Dependence ▪ Gas Limit and Loops ▪ DoS with (Unexpected) Throw ▪ DoS with Block Gas Limit ▪ Transaction-Ordering Dependence ▪ Style guide violation ▪ Costly Loop ▪ ERC20 API violation ▪ Unchecked external call ▪ Unchecked math ▪ Unsafe type inference ▪ Implicit visibility level ▪ Deployment Consistency ▪ Repository Consistency ▪ Data Consistency
Functional review	▪ Business Logics Review ▪ Functionality Checks ▪ Access Control & Authorization ▪ Escrow manipulation ▪ Token Supply manipulation ▪ Assets integrity ▪ User Balances manipulation ▪ Data Consistency manipulation ▪ Kill-Switch Mechanism ▪ Operation Trails & Event Generation

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

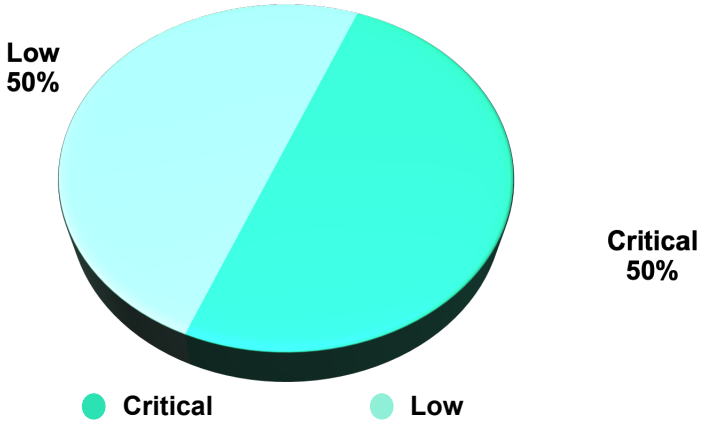
Executive Summary

According to the assessment, the Customer's smart contracts has some critical issues that should be fixed.

Our team performed an analysis of code functionality, manual audit, and automated checks with Mythril and Slither. All issues found during automated analysis were manually reviewed and important vulnerabilities are presented in the Audit overview section. A general overview is presented in AS-IS section and all found issues can be found in the Audit overview section.

During the audit we found 2 critical and 2 low severity issues and a bunch of code style issues.

Graph 1. The distribution of vulnerabilities.



Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to assets lose or data manipulations.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to assets lose or data manipulations.
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

AS-IS overview

Link.sol

Link is a basic implementation of ERC-20 Token. Link has following parameters:

- Name: Link
- Symbol: LINK
- Decimals: 18
- Total supply: 10m

Strong.sol

Strong is a basic implementation of ERC-20 Token. Link has following parameters:

- Name: Strong
- Symbol: STRONG
- Decimals: 18
- Total supply: 10m

PriceFeed.sol

PriceFeed is a contract to simulate price oracles. Used only for testing purposes.

sbCommunity.sol

sbCommunity contract is used to stake/unstake and to pay a reward to the sbController.

sbCommunity imports:

- **Ownable**
- **SafeMath** from *OpenZeppelin.*
- **IERC20** from *OpenZeppelin.*
- **sbTokensInterface**
- **sbControllerInterface**

- **sbStrongPoolInterface**
- **sbVotesInterface**

sbCommunity has 2 events:

- **NewPendingAdmin**
- **NewAdmin**

sbCommunity has 23 fields and constants:

- *bool internal initDone;*
- *address internal deployer;*
- *address internal constant ETH = address(0);*
- *string internal name;*
- *uint256 internal stakerRewardPercentage;*
- *IERC20 internal strongToken;*
- *sbTokensInterface internal sbTokens;*
- *sbControllerInterface internal sbController;*
- *sbStrongPoolInterface internal sbStrongPool;*
- *sbVotesInterface internal sbVotes;*
- *address internal sbTimelock;*
- *address internal admin;*
- *address internal pendingAdmin;*
- *mapping(address => mapping(address => uint256[])) internal stakerTokenStakeDays;*
- *mapping(address => mapping(address => uint256[])) internal stakerTokenStakeAmountsAdded;*
- *mapping(address => mapping(address => uint256[])) internal stakerTokenStakeAmountsSubtracted;*
- *mapping(address => uint256[]) internal tokenStakeDays;*

- *mapping(address => uint256[]) internal tokenStakeAmountsAdded;*
- *mapping(address => uint256[]) internal tokenStakeAmountsSubtracted;*
- *mapping(address => uint256) internal stakerDayLastClaimedFor;*
- *mapping(uint256 => uint256) internal dayServiceRewards;*
- *address[] internal services;*
- *mapping(address => string[]) internal serviceTags;*

sbCommunity has 44 functions:

- *init* – a public function used to initialize contract fields. Can only be called by the contract deployer.
- *setStakerRewardPercentage* – an external function used to set a staker reward percent. Can only be called by the sbTimelock contract.
- *getTokenStake* – an external view function used to fetch sum of specified tokens that has been staked during the specified day.
- *serviceAccepted* – an external view function used to check whether a specified service exists or no.
- *receiveRewards* – an external function used by the sbController contract to receive a reward.
- *setPendingAdmin* – a public function used to set up a new pending admin. Can only be called by the admin.
- *acceptAdmin* – a public function used to accept admin rights. Can only be called by the *pendingAdmin*.
- *getAdminAddressUsed* – a public view function used to fetch admin address.
- *getPendingAdminAddressUsed* – a public view function used to fetch the *pendingAdmin* address.
- *getSbControllerAddressUsed* – a public view function used to fetch the *sbController* address.
- *getStrongAddressUsed* – a public view function used to fetch the *Strong* token address.

- *getSbTokensAddressUsed* – a public view function used to fetch the *sbTokens* address.
- *getSbStrongPoolAddressUsed* – a public view function used to fetch the *sbStrongPool* address.
- getSbVotesAddressUsed – a public view function used to fetch the *sbVotes* address.
- *getSbTimelockAddressUsed* – a public view function used to fetch the *sbTimelock* address.
- *getDayServiceRewards* – a public view function used to fetch a reward for the service received at the specified day.
- *getName* – a public view function used to fetch the community name.
- *getCurrentDay* – a public view function used to fetch a current day.
- *getStakerRewardPercentage* – a public view function used to fetch a value of the *stakerRewardPercentage*.
- *stakeETH* – a public function used to stake ETH.
- *stakeERC20* – a public function used to stake ERC-20.
- *unstake* – a public function used to unstake tokens or ETH.
- *claimAll* – a public function used to claim Strong tokens for all days.
- *claimUpTo* – a public function used to claim tokens for up to specified day.
- *getRewardsDueAll* – a public view function used to calculate a reward that can be claimed by a specified address. Calculates a reward until current day.
- *getRewardsDueUpTo* – a public view function used to calculate a reward that can be claimed by a specified address. Calculates a reward until a specified day.
- *addService* – a public function used to add a new service. The service should have at least a minimum stake amount. Can only be called by the admin.

- *getServices* – a public view function used to fetch all services addresses.
- *getServiceTags* – a public view function used to fetch tags of a specified service.
- *addTag* – a public function used to add a new tag to a specified service. Can only be called by the admin.
- *getStakerDayLastClaimedFor* – a public view function used to fetch last claim day of a specified staker.
- *getStakerTokenStake* – a public view function used to fetch stake sum of a specified staker in a specified token at a specified day.
- *_getTokenStake* – an internal view function used to fetch sum of specified tokens that has been staked during the specified day.
- *_getStakerTokenStake* – an internal view function used to fetch stake sum of a specified staker in a specified token at a specified day.
- *_findAmount* – an internal pure function used to calculate staked sum at a specified day.
- *_getCurrentDay* – an internal view function used to fetch a current day.
- *_updateStakerTokenStake* – an internal function used to update staker balance during a stake.
- *_updateTokenStake* – an internal function used to update token balance during a stake.
- *_updateStake* – an internal function used to by the *_updateStakerTokenStake* and _updateTokenStake functions to update balances.
- *_serviceExists* – an internal view function used to check whether a service exists or no.
- *_serviceTagExists* – an internal view function used to check whether a service tag exists or no.
- *_claim* – an internal function used to claim tokens.
- *_getRewardsDue* – an internal view function used to calculate amount of tokens that can be claimed.

- *_getYearDayIsIn* – an internal view function used to calculate how may years are in specified days range.

sbController.sol

sbController is a contract used to manage communities and send rewards to them.

sbController imports:

- **SafeMath** from *OpenZeppelin*
- **IERC20** from *OpenZeppelin*
- **sbTokensInterface**
- **sbCommunityInterface**
- **sbStrongPoolInterface**

Contract ***sbController*** has 16 fields:

- *bool internal initDone;*
- *address internal deployer;*
- *address internal sbTimelock;*
- *IERC20 internal strongToken;*
- *sbTokensInterface internal sbTokens;*
- *sbStrongPoolInterface internal sbStrongPool;*
- *uint256 internal startDay;*
- *mapping(uint256 => uint256) internal COMMUNITY_DAILY_REWARDS_BY_YEAR;*
- *mapping(uint256 => uint256) internal STRONGPOOL_DAILY_REWARDS_BY_YEAR;*
- *uint256 internal MAX_YEARS;*
- *address[] internal communities;*
- *mapping(uint256 => uint256) internal dayStakeTotal;*

- *mapping(address => mapping(uint256 => uint256)) internal communityDayStake;*
- *mapping(address => mapping(uint256 => uint256)) internal communityDayRewards;*
- *mapping(address => uint256) internal communityDayStart;*
- *uint256 internal dayLastReleasedRewardsFor;*

sbController has 26 functions:

- *constructor* – sets up deployer address.
- *init* – an external function used to initialize contract fields.
- *upToDate* – an external view function. Checks whether a last reward release was yesterday.
- *addCommunity* – an external function used to add a new community. Can only be called by the *sbTimelock* contract.
- *getCommunities* – an external view function used to fetch all communities.
- *getDayStakeTotal* – an external view function used to fetch total sum that have been staked during a specified day. The sum is in the USD equivalence.
- *getCommunityDayStake* – an external view function used to fetch total sum staked by a specified community during a specified day. The sum is in the USD equivalence.
- *getCommunityDayRewards* – an external view function used to fetch a community reward at the specified day.
- *getCommunityDailyRewards* – an external view function used to fetch communities daily reward at the specified day.
- *getStrongPoolDailyRewards* – an external view function used to fetch the Strong Pool daily reward at the specified day.
- *getStartDay* – an external view function used to fetch the start day.

- *communityAccepted* – an external view function used to check whether a specified community exists or no.
- *getMaxYears* – a public view function used to fetch *MAX_YEARS* value.
- *getCommunityDayStart* – a public view function used to get a start day of the specified community.
- *getSbTimelockAddressUsed* – a public view function used to fetch an address of the *sbTimelock* contract.
- *getStrongAddressUsed* – a public view function used to fetch an address of the *strong token* contract.
- *getSbTokensAddressUsed* – a public view function used to fetch an address of the *sbTokens* contract.
- *getSbStrongPoolAddressUsed* – a public view function used to fetch an address of the *getSbTokensAddressUsed* contract.
- *getCurrentYear* –_a public view function used to fetch a current year of activity.
- *getYearDayIsIn* – a public view function used to calculate a number of years in a specified days.
- *getCurrentDay* – a public view function used to fetch a current day.
- *getDayLastReleasedRewardsFor* – a public view function used to fetch a latest day of reward release.
- *releaseRewards* – a public function used to pay a reward to communities and strong pool. The reward is paid in Strong tokens.
- *_getCurrentDay()* – an internal view function used to fetch a current day.
- *_communityExists* – an internal view function used to check whether a specified community exists or no.
- *_getYearDayIsIn* – an internal view function used to calculate a number of years in a specified number of days.

sbGovernor.sol

sbGovernor is a contract used to work with vote proposals.

sbGovernor imports:

- **sbVotesInterface**
- **sbTimelockInterface**

sbGovernor has 9 fields and constants:

- *string public constant name = 'sbGovernor';*
- *sbTimelockInterface public sbTimelock;*
- *sbVotesInterface public sbVotes;*
- *address public guardian;*
- *uint256 public proposalCount;*
- *mapping(uint256 => Proposal) public proposals;*
- *mapping(address => uint256) public latestProposalIds;*
- *bytes32 public constant DOMAIN_TYPEHASH = keccak256('EIP712Domain(string name,uint256 chainId,address verifyingContract)');*
- *bytes32 public constant BALLOT_TYPEHASH = keccak256('Ballot(uint256 proposalId,bool support)');*

Contract ***sbGovernor*** declares 2 data structures:

- *Proposal*
- *Receipt*

Contract ***sbGovernor*** has 1 enum:

- enum *ProposalState* { Pending, Active, Canceled, Defeated, Succeeded, Queued, Expired, Executed }

Contract ***sbGovernor*** declares 5 events:

- *ProposalCreated*
- *VoteCast*
- *ProposalCanceled*

- *ProposalQueued*
- *ProposalExecuted*

sbGovernor has 17 functions:

- *constructor* – used to set up *sbTimelockAddress, sbVotesAddress* and *guardian_* addresses.
- *propose* – a public function used to create a new proposal. The one who propose should have at least *proposalThreshold* tokens (at least 1% of total Strong supply).
- *queue* – a public function used to queue a successful proposal execution.
- *_queueOrRevert* – an internal function used to queue a proposal.
- *execute* – a public payable function used to execute queued proposal.
- *cancel* – a public function. Allows a guard to cancel any non-executed proposal.
- *getActions* – a public view function used to fetch actions of a specified proposal.
- *getReceipt* – a public view function used to fetch a vote result of a specified voter in a specified proposal.
- *state* – a public view function used to fetch state of a specified proposal.
- *castVote* – a public function used to vote for or against a specified proposal.
- *_castVote* – an internal function used to vote for or against a specified proposal.
- *__acceptAdmin* – a public function used to accept admin rules of the sbTimelock contract by the *sbController* contract. Can only be called by the guardian.
- *__abdicate* – a public function used to renounce guardian permissions. Can only be called by the guardian.

- *__queueSetTimelockPendingAdmin* – a public function used to queue a transaction to set a new pending admin on the sbTimelock contract. Can only be called by the guardian.
- *__executeSetTimelockPendingAdmin* – a public function used to execute a transaction to set a new pending admin on the sbTimelock contract. Can only be called by the guardian.
- *add256* and *sub256* – typical safemath functions.

sbStrongPool.sol

sbStrongPool is a contract that allows users to stake STRONG tokens. It also receives rewards from sbController if there are any STRONG tokens staked on a given day.

sbStrongPool imports:

- **SafeMath** from the OpenZeppelin
- **IERC20** from the OpenZeppelin
- **sbVotesInterface**
- **sbTimelockInterface**

sbStrongPool has 16 fields:

- *bool internal initDone;*
- *address internal deployer;*
- *IERC20 internal strongToken;*
- *sbControllerInterface internal sbController;*
- *sbVotesInterface internal sbVotes;*
- *address internal sbTimelock;*
- *uint256 internal minStake;*
- *mapping(address => uint256[]) internal stakerStakeDays;*
- *mapping(address => uint256[]) internal stakerStakeAmountsAdded;*
- *mapping(address => uint256[]) internal stakerStakeAmountsSubtracted;*

- *mapping(address => uint256) internal stakerIndexLastClaimedOn;*
- *uint256[] internal stakeDays;*
- *uint256[] internal stakeAmountsAdded;*
- *uint256[] internal stakeAmountsSubtracted;*
- *mapping(address => uint256) internal stakerDayLastClaimedFor;*
- *mapping(uint256 => uint256) internal dayRewards;*

sbStrongPool has 29 functions:

- *constructor* – sets message sender address as *deployer*.
- *init* – a public function used to set contract fields. Can only be called only once and only by the *deployer*.
- *updateMinStake* – an external function used to update minimum stake amount. Can only be called by the *sbTimelock* contract.
- *stakeFor* – an external function used to stake STRONG tokens on behalf of a specified address.
- *getStake* – an external view function used to fetch staked sum made at a specified day.
- *receiveRewards* – an external function used by the sbController to receive a reward.
- *getDayRewards* – a public view function used to get a reward for a specified day.
- *stake* – a public function used to stake STRONG tokens.
- *unstake* – a public function used to withdraw staked tokens.
- *getMinStakeAmount* – a public view function used to get minimum allowed stake amount.
- *getSbControllerAddressUsed* – a public view function used to get an address of the *sbController* contract.
- *getStrongAddressUsed* – a public view function used to get an address of the *STRONG* token contract.

- *getSbVotesAddressUsed* – a public view function used to get an address of the *sbVotes* contract.
- *getSbTimelockAddressUsed* – a public view function used to get an address of the *sbTimelock* contract.
- *getStakerDayLastClaimedFor* – a public view function used to get last claim date of a specified staker.
- *claimAll* – a public function used to claim all rewards until a current day.
- *claimUpTo* – a public function used to claim all rewards until a specified day.
- *getRewardsDueAll* – a public view function used to calculate currently available reward of a staker.
- *getRewardsDueUpTo* – a public view function used to calculate an available reward of a staker until a specified day.
- *getStakerStake* – an public view function used to get a stake sum of a staker at a specified day.
- *minStaked* – an external view function used to get a minimum stake sum of a specified staker.
- *_getStake* – an internal view function used to fetch a total staked sum made at a specified day.
- *_getStakerStake* – an internal view function used to get a stake sum of a staker at a specified day
- *_findAmount* – an internal pure function used to find staked amount at the specified day.
- *_getCurrentDay* – an internal view function used to get a current day.
- *_updateStake* – an internal function used to change stake sum.
- *_claim* – an internal function used to claim all rewards until a specified day.
- *_getRewardsDue* – an internal view function used to calculate an available reward of a staker until a specified day.

- *_getYearDayIsIn* – an internal pure function used to calculate how many years are in a specified date range.

sbTimelock.sol

sbTimelock is a contract used to queue, execute and cancel transactions.

sbTimelock imports:

- **SafeMath** from the OpenZeppelin

sbTimelock has 6 events:

- **NewAdmin**
- **NewPendingAdmin**
- **NewDelay**
- **CancelTransaction**
- **ExecuteTransaction**
- **QueueTransaction**

sbTimelock has 7 fields and constants:

- *uint256 public constant GRACE_PERIOD = 14 days;*
- *uint256 public constant MINIMUM_DELAY = 2 days;*
- *uint256 public constant MAXIMUM_DELAY = 30 days;*
- *address public admin;*
- *address public pendingAdmin;*
- *uint256 public delay;*
- *mapping(bytes32 => bool) public queuedTransactions;*

sbTimelock has 9 functions:

- *constructor* – sets admin address and delay period.
- *receive* – default receive function.

- *setDelay* – a public function used to set a *delay* value. Can only be called from the contract itself as a result of an executed transaction.
- *acceptAdmin* – a public function used by a pending admin to accept his admin rights.
- *setPendingAdmin* – a public function used to set a pending admin. Can only be called from the contract itself as a result of an executed transaction.
- *queueTransaction* – a public function used to queue a new transaction. Can only be called by the admin.
- *cancelTransaction* – a public function used to cancel a transaction. Can only be called by the admin.
- *executeTransaction* – a public payable function used to execute a transaction. Can only be called by the admin.
- *getBlockTimestamp* – an internal view function used to get a current block timestamp.

sbTokens.sol

sbTokens used to manage tokens and their prices in the StrongDefi system.

sbTokens imports:

- **SafeMath** from the OpenZeppelin
- **SafeCast** from the OpenZeppelin
- **AggregatorV3Interface** from the ChainLink

sbTokens has 10 fields and constants:

- *bool internal initDone;*
- *address internal deployer;*
- *address internal sbTimelock;*
- *address[] internal tokens;*
- *address[] internal oracles;*
- *mapping(address => AggregatorV3Interface) internal priceFeeds;*

- *mapping(address => mapping(uint256 => uint256)) internal tokenDayPrice;*
- *mapping(address => uint80) internal tokenRoundLatest;*
- *mapping(address => uint256) internal tokenDayStart;*
- *uint256 internal dayLastRecordedPricesFor;*

sbTokens has 21 functions:

- *constructor* – sets deployer address.
- *init* – a public function used to set contract fields. Can only be called only once and only by the *deployer*.
- *upToDate* – an external view function used to check whether tokens prices are up to date.
- *addToken* – an external function used to add a new token and its price oracle. Can only be called from the *sbTimelock* contract.
- *getTokens* – an external view function used to fetch all tokens.
- *getOracles* – an external view function used to fetch all price oracles.
- *getTokenPrices* – an external view function used to fetch all token prices.
- *tokenAccepted* – an external view function used to check whether a specified token exists in the system or no.
- *getTokenPrice* – an external view function used to fetch a specified token price at a specified day.
- *getDayLastRecordedPricesFor* – a public view function used to get a value of the dayLastRecordedPricesFor field.
- *getSbTimelockAddressUsed* – a public view function used to get an address of the *sbTimelock* contract.
- *getTokenRoundLatest* – a public view function used to fetch a latest round of a specified token.
- *getTokenDayStart* – a public view function used to fetch a day a specified token been added.

- *getCurrentDay* – a public view function used to fetch a current day.
- *recordTokenPrices* – a public function used to record latest tokens prices.
- *_getDayClosingPrice* – an internal function used to get a specified token price at a specified day.
- *_getRoundBeforeTimestamp* – an internal function used to get round before a specified timestamp.
- *_getCurrentDay* – an internal view function used to get a current day.
- *_dayToTimestamp* – an internal pure function used to convert days to timestamp.
- *_tokenExists* – an internal view function used to check whether a specified token exist.
- *_oracleExists* – an internal view function used to check whether a specified oracle exist.

sbVotes.sol

sbVotes contract is used to manage votes and pay rewards.

sbVotes contract has 5 imports:

- **SafeMath** — from OpenZeppelin
- **IERC20** — from OpenZeppelin
- **sbControllerInterface**
- **sbStrongPoolInterface**
- **sbCommunityInterface**

sbVotes contract has 2 data structures:

- *Checkpoint*
- *Vote*

sbVotes contract has 19 fields:

- *bool internal initDone;*
- *address internal deployer;*
- *sbControllerInterface internal sbController;*

- *sbStrongPoolInterface internal sbStrongPool;*
- *IERC20 internal strongToken;*
- *string public constant name;*
- *mapping(address => uint96) internal balances;*
- *mapping(address => address) public delegates;*
- *mapping(address => mapping(uint32 => Checkpoint));*
- *mapping(address => uint32) public numCheckpoints;*
- *mapping(address => uint256) public nonces;*
- *mapping(address => mapping(address => address[])) internal voterCommunityServices — a list of voter community services;*
- *mapping(address => mapping(address => mapping(address => uint256))) internal voterCommunityServiceVotes;*
- *mapping(address => uint256) internal voterVotesOut;*
- *mapping(address => mapping(address => Vote[])) internal serviceCommunityVotes;*
- *mapping(address => mapping(address => uint256)) internal serviceCommunityIndexLastClaimedOn;*
- *mapping(address => Vote[]) internal communityVotes;*
- *mapping(address => uint256) internal serviceDayLastClaimedFor;*
- *mapping(address => mapping(uint256 => uint256)) internal communityDayRewards;*

sbVotes has 48 functions:

- *constructor* — Sets deployer.
- *init* — a public function that initializes contract fields.
- *updateVotes* — an external function used to update votes number.
- *getCurrentProposalVotes* — an external view function used to fetch current proposal votes for an account.

- *getPriorProposalVotes* — an external view function used to get prior proposal votes for an account.
- *getCommunityVote* — an external view function used to get community votes.
- *receiveRewards* — an external function used to receive a reward.
- *getServiceDayLastClaimedFor* — a public view function used to get the last day when a specified service claimed a reward.
- *getSbControllerAddressUsed* — a public view function used to get the *sbController* contract address.
- *getSbStrongPoolAddressUsed* — a public view function used to get the *sbStrongPool* contract address.
- *getStrongAddressUsed* — a public view function used to get the *STRONG token* contract address.
- *getCommunityDayRewards* — a public view function used to calculate a reward sum at a specified day.
- *recallAllVotes* — a public function used to recall all votes.
- *delegate* — a public function used to delegate voting rights.
- *getAvailableServiceVotes* — a public view function used to get available service votes.
- *getVoterCommunityServices* — public view function used to get a voter community services.
- *getVoterCommunityServiceVotes* — a public view used to get votes of a voter community services.
- *vote* — a public function used for voting.
- *recallVote* — a public function used to recall vote.
- *claimAll* — a public function used to claim all rewards until current day.
- *claimUpTo* — a public function used to claim all rewards until a specified day.

- *getRewardsDueAll* — a public view function used calculate a reward sum until a current day.
- *getRewardsDueUpTo* — a public view function used calculate a reward sum until a specified day.
- *getServiceCommunityVote* – a public view function used to fetch votes of a community service at a specified day.
- *_getCommunityVote* — an internal view function used to get community votes.
- *_getServiceCommunityVote* — an internal view function used to get votes of a specified service with a specified community.
- _findAmount — an internal pure function used to calculate votes.
- *_getCurrentDay* — an internal view function used to get a current day.
- *_updateServiceCommunityVotes* — an internal function used to update service community votes.
- *_updateCommunityVotes* — an internal function used to update community votes.
- *_updateVotes* — an internal function used to update votes.
- *_addVotes* — an internal function used to add votes.
- *_subbVotes* — an internal function used to sub votes.
- *_addDelegates* — an internal function used to add delegate votes.
- *_subtactDelegates* — an internal function used to subtract delegated votes.
- *_delegate* — an internal function used to delegate votes.
- *_moveDelegates* — an internal function used to move delegates.
- *_writeCheckpoint* — an internal function used to write checkpoint.
- *_safe32* — an internal pure function that checks if number is safe 32.
- *_safe96* — an internal pure function that checks if number is safe 96.
- *_add96* — an internal pure function used to safe add numbers.

- *_sub96* — an internal pure function used to safe subtract.
- *_getCurrentProposalVotes* — an internal view function used to get current proposal votes for an account.
- *_getAvailableServiceVotes* — an internal view function used to get available votes of a service.
- *_voterCommunityServiceExists* — an internal view function used to check whether voter community service exists.
- *_recallAllVotes* — internal function used to recall all votes.
- *_claim* — an internal function used to claim all rewards until a specified day.
- *_getRewardsDue* — an internal view function used to calculate an available reward of a staker until a specified day.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Audit overview

Critical

1. *stakeERC20* function of the *sbCommunity* contract should validate tokens transfer result. It's not recommended to rely on exceptions of the *transferFrom* function because some ERC-20 implementation may not produce them.

2. *stakeFor* and *stake* functions of the *sbStrongPool* contract should validate tokens transfer result. It's not recommended to rely on exceptions of the *transferFrom* function because some ERC-20 implementation may not produce them.

   StrongBlock Response:

   We will review the possibility of intending to support an ERC20 token that does not follow the SHOULD specification for failing on an invalid transfer or transferFrom, and if we find that supporting such tokens may be desired then we will validate the results of those functions.

   Hacken Response:

   According to the ERC-20 specification, the transfer function must throw an exception. But it's impossible to enforce such implementation. List of buggy contracts: https://github.com/sec-bit/awesome-buggy-erc20-tokens/blob/master/csv/verify-reverse-in-transferFrom.o.csv

   As soon as it's possible to add new tokens into the system via voting it's better to validate transfer results or to check each proposed token manually and to cancel such proposals.

High

No high severity issues have been found.

Medium

No medium severity issues have been found.

Low

1. Move common code of _getRewardsDue and _claim functions to a separate function instead of code duplication during the reward calculation.
2. *nonces* and *serviceCommunityIndexLastClaimedOn* fields of the *sbVotes* contract are never used and should be removed.

   StrongBlock Response:

   Code that is reasonably common, and not duplicated for any good reason, will be addressed and streamlined to avoid possibility of conflating logical constructs in a way that could lead to implementation errors.

   Hacken Response:

   Low and lowest severity issues are advisory and may be left unfixed.

Lowest / Code style / Best Practice

1. sbGovernance contains 2 commented out function. Remove them to clean up the code.
2. sbGovernance implements its own safemath functions. Consider using SafeMath library of the OpenZeppelin.
3. *stakeFor* and *stake* of the sbStrongPool contract has almost similar content and can be merged to reduce code duplication.

   StrongBlock Response:

   Code that is reasonably common, and not duplicated for any good reason, will be addressed and streamlined to avoid possibility of conflating logical constructs in a way that could lead to implementation errors.

   Hacken Response:

   Low and lowest severity issues are advisory and may be left unfixed.

Conclusion

Smart contracts within the scope was manually reviewed and analyzed with static analysis tools. For the contract high level description of functionality was presented in As-is overview section of the report.

Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security engineers found 2 critical, 2 low and 3 lowest severity issues during audit. It’s recommended to fix all critical issues.

As Smart Contract Auditor we have concerns for next items:

Category	Check Item	Comment
Functional review	■ Assets integrity	Always validate transfer results when working with untrusted erc-20 implementations.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed in accordance with the best industry practices at the date of this report, in relation to cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

The audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.

Technical Disclaimer

Smart contracts are deployed and executed on blockchain platform. The platform, its programming language, and other software related to the smart contract can have own vulnerabilities that can lead to hacks. Thus, the audit can’t guarantee explicit security of the audited smart contracts.