HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Axion
Date: September 3rd, 2020

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

Document

Name	Smart Contract Code Review and Security Analysis Report for Axion
Type	Token, auction, tokenswap, staking
Platform	Ethereum / Solidity
Methods	Architecture Review, Functional Testing, Computer-Aided Verification, Manual Review
Repository	https://github.com/Rock-n-Block/axion-contracts
Commit	87afb419b029ea4b58508fd59a4f6163f94c61da
Timeline	AUGUST 31, 2020 - SEPTEMBER 03, 2020
Changelog	03RD SEP 2020 - Initial Audit

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 Axion (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 August 31st, 2020 - September 03rd, 2020.

Scope

The scope of the project is smart contracts in the repository:

Audit Repository	
https://github.com/Rock-n-Block/axion-contracts	
Commit	87afb419b029ea4b58508fd59a4f6163f94c61da

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 ▪ Data Consistency ▪ Deployment Consistency ▪ Repository Consistency
Functional review	▪ Business Logics Review ▪ Functionality Checks ▪ Access Control & Authorization ▪ Escrow manipulation ▪ Token Supply manipulation ▪ 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, Customer`s smart contracts have high and medium vulnerabilities that should be fixed.

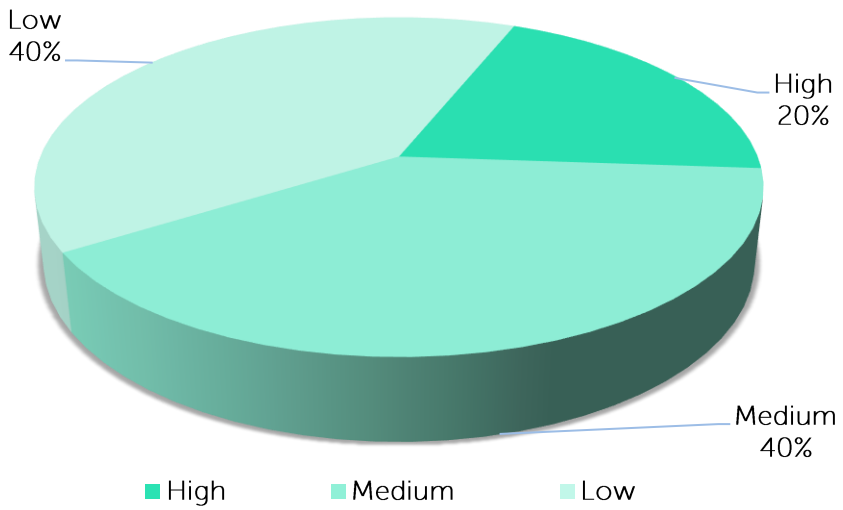
Insecure	Poor secured	Secured	Well-secured

You are here

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 3 high, 6 medium, and 6 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
High	Critical vulnerabilities are usually straightforward to exploit and can lead to assets loss or data manipulations.
Medium	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g., public access to crucial functions
Low	Medium-level vulnerabilities are important to fix; however, they can't lead to assets loss or data manipulations.
Informational	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets, that can't have a 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

Token.sol

Token imports ERC20.sol, AccessControl.sol, SafeMath.sol from OpenZeppelin, and IToken.sol from the project.

Token inherits IToken, ERC20, and AccessControl.

Token is a basic ERC20 token with the possibility to swap from another address. Swap is allowed only from a single address with a setter role. If the swap is finished, there will be 6 addresses with the possibility to mint any sum of tokens.

Contract Token has 5 fields and constants:

- *bytes32 private constant MINTER_ROLE = keccak256("MINTER_ROLE");*
- *bytes32 private constant SETTER_ROLE = keccak256("SETTER_ROLE");*
- *address private swapToken* - stores an address of a token that will be swapped.
- *bool private swapIsOver* - indicates whether swap is finished or no.
- *mapping(address => uint256) private swapTokenBalanceOf* - stores swap balances of specified addresses.

Contract Token has 2 modifiers:

- *onlyMinter*() - checks if the function caller has a minter role.
- *onlySetter*()- checks if the function caller has a setter role.

Contract Token has 12 functions:

- constructor - sets token name, symbols, address of a token to swap from and an address with the setter role.
- *init* - an external function that can only be called by the setter. Used to specify a list of addresses with the minter role and to finish swap process.
- *getMinterRole* - an external pure function that returns a value used as a key for the minter role.
- *getSetterRole* - an external pure function that returns a value used as a key for the setter role.
- *getSwapTOken* - an external pure function that returns the swap token address.
- *getSwapTokenBalanceOf* - an external view function used to get balance of swap tokens of a specified address.
- *initDeposit* - an external function that can only be called by the setter. Used to deposit tokens for swap and to increase *swapTokenBalance* of a caller.
- *initWithdraw* - an external function that can only be called by the setter. Used to withdraw swap tokens and to decrease *swapTokenBalance* of a caller.
- *initSwap* - an external function that can only be called by the setter. Used to swap all tokens of a caller. As a result, the caller will receive new tokens in 1 to 1 proportion with swap tokens.
- *mint* - an external function that can only be called by the minter. Used to mint tokens.
- *burn* - an external function that can only be called by the minter. Used to mint tokens.
- *getNow* - an external view function used to fetch the current timestamp.

Auction.sol

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

Auction imports IERC20.sol, AccessControl.sol, SafeMath.sol from OpenZeppelin and IToken.sol, IUniswapV2Router02.sol, IAuction.sol from the project.

Auction inherits *IAuction* and *AccessControl*.

Auction is a contract that allows one to make a bet in ETH and to withdraw this bet in Tokens. In a case when a ref provided during the bet, referral bonuses are received by both sides. The contract works as an exchange with max exchange rate obtained from the uniswap.

Contract Auction has 15 fields and constants:

- *bytes32 public constant MANAGER_ROLE = keccak256("MANAGER_ROLE");*
- *bytes32 public constant CALLER_ROLE = keccak256("CALLER_ROLE");*
- *mapping(uint256 => AuctionReserves) public reservesOf* - stores reserve sum of Auctions.
- *mapping(address => uint256[]) public auctionsOf* - stores auction id's where an address owner participated at.
- *mapping(uint256 => mapping(address => UserBet)) public auctionBetOf* - stores auction bets of an address owner.
- *mapping(uint256 => mapping(address => bool)) public existAuctionsOf* - identifies whether an address participated in an auction or no.
- *uint256 public start* - the contract deploy timestamp.
- *uint256 public currentAuctionId* - a current auction id.
- *uint256 public stepTimestamp* - delay between auction steps.
- *uint256 public uniswapPercent* - bonus for swapping with uniswap.
- *address public mainToken* - token contract address.
- *address public staking* - staking contract address. This address is used to send tokens from uniswap
- *address payable public uniswap* - uniswap contract address.
- *address payable public recipient* - address of the wallet where ETH is transferred during betting.

- *bool public init_* - stores init status.

Contract Auction has 2 modifiers:

- *onlyCaller*() - checks if the function caller has caller role.
- onlyManager()- checks if the function caller has manager role.

Contract Auction has 2 data structures:

- *AuctionReserves* - used to store reserves of auction step.
- *UserBet* - used to store a user bet for an auction step.

Contract Auction has 15 functions:

- *constructor* - sets *init_* to false.
- *init* - an external function used to initialize values of *start*, *stepTimestamp*, *uniswapPercent*, *mainToken*, *staking*, *uniswap* and *recipient* fields. The function sets *MANAGER_ROLE* to the *_manager* address and *CALLER_ROLE* to the *_nativeSwap*, *_foreignSwap* and *_staking* addresses. Can be called only once.
- auctionsOf - a public view function used to fetch all auctions where a specified address participated at.
- setUniswapPercent - an external function used to set uniswap percent value. Can be executed only from an address with the manager role.
- *bet* - an external payable function used to make a bet in the latest auction.
- *withdraw* - an external function used to withdraw tokens after an auction step end.
- callIncomeDailyTokensTrigger - an external function used to set a reserve tokens sum of a current auction. Can be executed only from an address with the caller role.
- callIncomeWeeklyTokensTrigger - an external function used to set a reserve tokens sum of a weekly auction. Can be executed only from an address with the caller role.
- calculateNearestWeeklyAuction - a public view function used to calculate a weekly auction step.
- *calculateStepsFromStart* - a public view function used to calculate a current auction step.

- *_changePayoutWithUniswap* - an internal view function used to calculate payout sum. In a case when the payout sum is greater than a value received from the uniswap, uniswap value will be returned as a result.
- *_calculatePayout* - an internal function used to calculate a payout for a specified auction and sum.
- *_calculateRecipientAndUniswapAmountsToSend* - a private function used to calculate sums of ETH that goes to the *recipient* and *uniswap* addresses during the betting process. The *recipient* receives 20% the sum, the *uniswap* receives 80% of the sum.
- *_calculateRefAndUserAmountsToMint* - a private pure function used to calculate referral bonuses during the withdrawal process. A referrer receives 10% of extra tokens, and a user receives 20% of extra tokens.
- *_swapEth* - a private function used to send ETH to the uniswap wallet during betting.

NativeSwap.sol

NativeSwap imports IERC20.sol, SafeMath.sol from OpenZeppelin and IToken.sol, IAuction.sol from the project.

NativeSwap is a contract that allows swapping tokens. The sum of the main token depends on steps passed from the contract deploys date.

Contract NativeSwap has 7 fields and constants:

- *uint256 private start* - the contract deploy timestamp.
- *uint256 private stepTimestamp* - delay between auction steps.
- address private swapToken - an address of a token to swap from.
- address private mainToken - an address of the main token.
- address private dailyAuction - an address of the Auction contract.
- *bool public init_* - stores init status.
- *mapping(address => uint256) private swapTokenBalanceOf* - stores balances of swap token of specified address.

Contract NativeSwap has 13 functions:

- *constructor* - sets *init_* to false.
- *init* - an external function used to set values of *stepTimestamp*, *swapToken*, *mainToken*, *dailyAuction* and *start* fields. Can be executed only once.
- getStart - an external view function used to fetch the contract init timestamp.
- getStepTimestamp - an external view function used to fetch the *stepTimestamp* value.
- *getSwapToken* - an external view function used to fetch the swap token address.
- *getMainToken* - an external view function used to fetch the main token address.
- *getDailyAuction* - an external view function used to fetch the auction address.
- getSwapTokenBalanceOf - an external view function used to fetch balance of the swap token of a specified address.
- *deposit* - an external function used to deposit swap tokens. Should be called only if an allowance is set on the swap token contract.
- *withdraw* - an external function used to withdraw swap tokens.
- *swapNativeToken* - an external function used to swap tokens of a function caller.
- *readSwapNativeToken* - an external view function used to calculate swap sum.
- *_calculateDeltaPenalty* - an internal view function used to calculate swap penalty.

Staking.sol

Staking imports IERC20.sol, AccessControl.sol, SafeMath.sol from OpenZeppelin and IToken.sol, IStaking.sol, IAuction.sol, ISubBalances.sol from the project.

Staking inherits *IStaking* and *AccessControl*.

Staking is a contract used to stake tokens. Staking rates are calculating depending on the time passed after a stake start.

Contract Staking has 15 fields and constants:

- *uint256 private _sessionsIds* - total number of stakes.

- *bytes32 public constant EXTERNAL_STAKER_ROLE = keccak256("EXTERNAL_STAKER_ROLE");*
- *address public mainToken* - token address.
- *address public auction* - auction address.
- *address public subBalances* - address of the ISubBalances contract.
- *uint256 public shareRate* - share rate. Used to calculate shares amount.
- *uint256 public sharesTotalSupply* - total shares.
- *uint256 public lastMainTokenBalance* - snapshot of the token total supply on a time of last *payout* function call.
- uint256 public nextPayoutCall - time when a next payout allowed.
- uint256 public stepTimestamp - delay between steps.
- uint256 public startContract - contract deploy timestamp.
- *bool public init_* - stores init status.
- mapping(address => mapping(uint256 => Session)) public sessionDataOf - stakes storage.
- mapping(address => uint256[]) public sessionsOf - stakes of specific address.
- Payout[] public payouts - list of payouts.

Contract Staking has 2 modifiers:

- *onlyExternalStaker* () - checks if the function caller has external staker role.

Contract Staking has 2 data structures:

- *Payout* - used to store payout information.
- *Session* - used to store stake information.

Contract Staking has 15 functions:

- *constructor* - sets *init_* to false.
- *init* - an external function used to initialize values of *mainToken*, *auction*, *subBalances*, *shareRate*, *lastMainTokenBalance*, *stepTimestamp*, *nextPayoutCall* and

startContract fields. The function sets *EXTERNAL_STAKER_ROLE* to the *_externalStaker* address. Can be called only once.

- sessionsOf - an external view function used to fetch all stakes of a specified address.
- stake - an external function used to stake tokens.
- externalStake - an external function used to stake tokens on behalf of a specified address. Can be executed only by an address with the external staker role.
- *unstake* - an external function used to finish specified staking of a message sender. There're 4 options:
  - early - finish stake with an early penalty.
  - in time - finish stake with bonus. Applies on a date of the stake end plus 13 days.
  - late - finish stake with a late penalty. Applies between 14 and 713 days after a stake end.
  - Nothing - applies after 713 days of staking end date.
- *readUnstake* - an external view function used to calculate an unstake sum of a specified account and staking id.
- *makePayout* - an external function used to initiate new payout.
- *readPayout* - an external view function used to calculate a payout sum.
- *_getPayout* - an internal function used to process payout. Tokens are minted for the contract address during the payout process.
- *_getStakersSharesAmount* - an internal view function used to calculate staker shares sum.
- *_getShareRate* - an internal view function used to fetch stake share rate.
- *getNow0x* - an external view function used to fetch a current block timestamp.

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

Audit overview

Critical

High

1. To avoid *Token* total supply manipulation, it's recommended to limit the number of addresses with a minter role. Currently, the init function of the Token contract expects 6 addresses that will have this role, whether the repository have only 3 contracts that requires the minter role.
2. The externalStake function of the Staking contract can perform stake for any account without approval. Such behavior can not consider as safe because tokens are burned from the account.
3. The *unstake* function of the *Staking* contract has reentrancy vulnerability. It's recommended to store *shares* as local variable and set *sessionDataOf[msg.sender][sessionId].shares = 0* in the beginning of the function.

Medium

1. It's recommended to move a value of the *stepTimestamp* to constant because it's implied that its value should always be equal to 1 day.
2. It's recommended to validate a result value of the *transferFrom* function in the *deposit* function of the *NativeSwap* contract.
3. The *_getShareRate* function of the *Staking* contract uses hardcoded value of the token decimals. This value should be fetched from the token contract.
4. Functions *unstake* and *readUnstake* of the *Staking* contract are overcomplicated and should be split to separate functions.
5. *_getShareRate* and *_getStakersSharesAmount* function of the *Staking* contract has "magic numbers" that should be moved to a named constants.
6. The *unstake* function of the *Staking* contract should have if - else statements instead of simple if statements.

Low

1. To avoid extra type casts, field swapToken of the Token contract can be specified as IERC20 instead of address.
2. As soon as all swap functions of the *Token* contract can only be called from the setter address, swap balance can be stored as *uint256* type instead of *mapping* type.
3. Validation should be moved out of the cycle in the *init* function of the *Token* contract.
4. To decrease code duplication, the *swapNativeToken* function of the *NativeSwap* contract should use *readSwapNativeToken* function to get swap sum.
5. The *readUnstake* function of the Staking contact does not have unconditional return statement. It's recommended to add the default one.
6. *stake* and *externalStake* functions of the Staking contract have common code that can be moved to separate function to decrease code duplication.

Lowest / Code style / Best Practice

1. Function *_daysFromDate*, timestampFromDateTime, *_daysToDate* and *timestampToDate* have a lot of "magic" numbers that should be moved to named constants.
2. Function *getCirculation* does not return any result. *updateCirculationValue* is more suitable name.
3. A lot of other code-style issues can be found by running any static code analyzer specified in the Executive Summary section.

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

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 3 high, 6 medium, and 6 low severity issues during audit. High severity issues can lead to token supply manipulations or to unwilling funds lock. It’s recommended to fix all those issues.

Category	Check Item
Code review	▪ Style guide violation ▪ Data Consistency
Functional review	▪ Token Supply manipulation ▪ User Balances manipulation

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 the 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 the 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.