SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: AutoMiningToken
Date: 20 September, 2023

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 AutoMiningToken
Approved By	Paul Fomichov | Lead Solidity SC Auditor at Hacken OÜ
Tags	ERC20 token; Liquidity Pool
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://www.autominingtoken.com/
Changelog	15.06.2023 - Initial Review 01.08.2023 - Second Review 29.08.2023 - Third Review 20.09.2023 - Fourth Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by AutoMiningToken (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

AutoMiningToken is a platform that allows, through its token - AMT, investment in BTC mining projects. Users who own AMT will be able to profit from the returns generated by the project based on the amount of AMT they own on all circulating tokens.

- *AMT* — a simple ERC-20 token that does not mint any initial supply, but additional minting is allowed. Total supply is capped to 100 million tokens.
  It has the following attributes:
  - Name: AutoMiningToken
  - Symbol: AMT
  - Decimals: 18
  - Total supply: 100m tokens.
- *LiquidityAmt* — a simple ERC-20 token to be used for liquidity purposes that does not mint any initial supply, but additional minting is allowed.
  It has the following attributes:
  - Name: liqAutoMiningToken
  - Symbol: liqAMT
  - Decimals: 18
  - Total supply: Infinitive.
- Market — a contract that allows users to withdraw USDT tokens against their AMT balance (a sort of liquidation operation), as well as buy AMT with USDT tokens.
- Master — a contract that manages the liquidity pools and distributes rewards for liquidity providers and AMT token holders.
  Each reward charge is tracked through a snapshot mechanism. When the owner calls the "payRent" function, the reward for that specific moment will be distributed based on the snapshot of how many tokens they currently hold or how much they have utilized for the liquidity pool. **So, the total reward amount and the timing of its distribution will depend on the system owner.**
- BurnVault — a contract that allows users to burn their AMT tokens in exchange for some amount of BTCB balance in the contract.

Privileged roles

- The owner of the *AMT* contract can:

    - take snapshots
    - mint tokens
- The owner of the *LIQUIDITYAMT* contract can:
    - take snapshots
    - mint tokens
- The owner of the marketvault can:
    - set the Master contract address
    - can transfer the backing tokens to itself
- The owner of Master can:
    - can extend the approval amount for addrRouter
    - set the payer wallet
    - add liquidity to be locked for 2 years
    - mint AMT tokens

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are provided.
- Technical description is provided.
- NatSpec is sufficient.

Code quality

The total Code Quality score is **9** out of **10**.

- Development environment is configured.
- Solidity Style Guides are not followed.

Test coverage

Code coverage of the project is **100%** (branch coverage).

Security score

As a result of the audit, the code contains **no** issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **9.8**. The system users should acknowledge all the risks summed up in the risks section of the report.

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
15 June 2023	4	4	0	1
01 August 2023	2	2	2	0
29 August 2023	0	1	1	0
20 September 2023	0	0	0	0

Risks

- The system owner can **mint an infinite amount of *liqAMT* tokens**.
- In the Market contract, AMT price to buy with USDT is specified by a **centralized mechanism** and can be changed by the system owner.

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items considered:

Item	Description	Status	Related Issues
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Failed	I10
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Not Relevant	
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.	Passed	
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed	
Unchecked Call Return Value	The return value of a message call should be checked.	Passed	
Access Control & Authorization	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed	
SELFDESTRUCT Instruction	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant	
Check-Effect-Interaction	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed	
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed	
Deprecated Solidity Functions	Deprecated built-in functions should never be used.	Passed	
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.	Not Relevant	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	

Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Passed	
Authorization through tx.origin	tx.origin should not be used for authorization.	Not Relevant	
Block values as a proxy for time	Block numbers should not be used for time calculations.	Not Relevant	
Signature Unique Id	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Not Relevant	
Shadowing State Variable	State variables should not be shadowed.	Passed	
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant	
Incorrect Inheritance Order	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Passed	
Calls Only to Trusted Addresses	All external calls should be performed only to trusted addresses.	Passed	
Presence of Unused Variables	The code should not contain unused variables if this is not justified by design.	Passed	
EIP Standards Violation	EIP standards should not be violated.	Passed	
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed	
User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Passed	
Data Consistency	Smart contract data should be consistent all over the data flow.	Passed	

Flashloan Attack	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used. Contracts shouldn’t rely on values that can be changed in the same transaction.	Passed	
Token Supply Manipulation	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Passed	
Gas Limit and Loops	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Not Relevant	
Style Guide Violation	Style guides and best practices should be followed.	Failed	I09
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Passed	
Environment Consistency	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed	
Secure Oracles Usage	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Not Relevant	
Tests Coverage	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed	
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Passed	

Findings

■■■■ Critical

C01. Invalid Calculations

Impact	High
Likelihood	High

The *backingWithdrawl()* function implementation first burns the refunded tokens and then calculates the back rate for the backing token.

This order of functions called *burnFrom()->getBackRate()* is incorrect as the back rate is dependent on the total supply of the refunded token.

This will lead to users to get inconsistent and unfair amounts of backing tokens depending on the amount they refund.

Path:
./marketvault.sol : backingWithdrawl();

Recommendation: Follow the correct order in the function. First calculate the back rate based on the current total supply and then burn the tokens.

Found in: e0fe204

Status: Fixed (Revised commit: 0fd2faa)

■■■ High

H01. Contradiction in Docs

Impact	Low
Likelihood	High

Although it’s stated that “This contract allows for the buying and selling of AMT tokens with USDT and BTCb” in the *Market* contract, functions only support buying with *USDT* token.

Path:
./Production/contracts/Market.sol : buy();

Recommendation: Implement the mentioned requirement or update the explanation in the code.

Found in: 0fd2faa

Status: Fixed (Revised commit: b2516c5)

H02. Invalid Calculations

Impact	Medium
Likelihood	High

The *chargeFromTo()* function updates the values of *chargedAt* and *amtUsedAt* mappings incorrectly.

They need to be cumulatively updated by aggregating the charge amount of each user for the respective snapshot ID. However, in the function mentioned, it is updated for only one user (specifically, the 'msg.sender' of the function in our case), while the charges of other users are overlooked.

Path:
./Master.sol : chargeFromTo()

Recommendation: Update the lines as below;

```
chargedAt[i] += paidAti;
amtUsedAt[i] += amt.balanceOfAt(msg.sender, i);
```

Found in: b2516c5

Status: Fixed (Revised commit: c674d1f)

Medium

M01. Missing Validation

Impact	Medium
Likelihood	High

The *getBackRate()* function does not check if the *backingCoin* balance is greater than 0.

This can lead to situations where division by 0 is performed.

Path:
./marketvault.sol : getBackRate();

Recommendation: Add a validation check that the *backingCoin* balance is greater than 0.

Found in: e0fe204

Status: Fixed (Revised commit: 0fd2faa) (*getBackRate* function is removed)

M02. Best Practice Violation / Unchecked Return Value

Impact	Low
Likelihood	High

Return values of the *transfer* and *transferFrom* functions are not validated.

In the absence of proper validation, a situation might arise where function returns an error, yet the transaction still proceeds to completion.

This may lead to unexpected behavior.

Paths:
./marketvault.sol : backingWithdrawl(), buy(), charge();
./Market.sol : release(), payRent(), charge(), liqCharge(), addLiquidityLocking(), addLiquidity(), removeLiquidity();

Recommendation: Incorporate a check for the return value of the function to ensure it executes correctly. Or use Openzeppellins's SafeERC20.

Found in: e0fe204

Status: Fixed (Revised commit: c674d1f)

M03. Missing Event Emitting

Impact	Low
Likelihood	High

The *extendApprove*, *setPayerWallet*, *payRent*, *addLiquidityLocking*, *addLiquidity*, *removeLiquidity*, *mintMaster* functions in the *Master* contract do not emit an event.

The *backingWithdrawl*, *buy*, *setMaster* and *charge* functions of the *mastervault* contract do not emit an event.

Important state changes should emit events to allow users to track transactions on the front-end.

Paths:
./Master.sol : extendApprove(), setPayerWallet(), payRent(), addLiquidityLocking(), addLiquidity(), removeLiquidity(), mintMaster();
./mastervault : backingWithdrawl(), buy(), setMaster() , charge();

Recommendation: Emit the event whenever corresponding action happens.

Found in: e0fe204

Status: Fixed (Revised commit: 0fd2faa)

M04. Highly Permissive Role Access

Impact	High
Likelihood	Low

The owner of the *marketvault* contract can change the *Master* contract address at any moment and call the *charge()* function.

This can be used to withdraw all the *backingCoin* tokens from the contract if the changed *Master* contract is malicious, by setting the amount returned from the *Master.charge()* function as the maximum amount that can be withdrawn.

If a key leak were to occur, the potential consequences could be significant, potentially leading to security breaches and undermining the overall integrity of the system.

Path:
./marketvault.sol : setMaster(), charge();

Recommendation: It is recommended to restrict the scope of permissions for those roles.

To ensure transparency and accountability, it is advised to provide a comprehensive explanation of highly-permissive access in the system's public documentation. This would help to ensure that users are fully informed of the implications of such access and can make informed decisions accordingly.

Either restrict the owner's access to not to withdraw the funds or make the Master contract immutable to protect users’ funds.

Found in: e0fe204

Status: Fixed (Revised commit: 0fd2faa)

M05. Missing Validation

Impact	Low
Likelihood	High

backingWithdraw function does not check if the BTCB balance of the contract is not zero. In a scenario like mentioned, users will not get any BTCB in exchange for burning their AMT tokens.

Path:
./Production/contracts/BurnVault.sol : backingWithdraw();

Recommendation: Add a validation that checks that the BTCB balance is greater than 0.

Found in: 0fd2faa

Status: Fixed (Revised commit: b2516c5)

M06. Accumulation of Dust Values

Impact	Low
Likelihood	High

In the *Master* contract, the *charge* function distributes the rewards that users earned. The calculation is performed by dividing the total reward among all token holders, which may not always result in whole numbers. As a consequence, there might be fractional values, leading to the accumulation of dust.
A contract should not accumulate dust values left after swaps or reward distributions.

Path:
./Production/contracts/Master.sol : charge(), chargeFromTo();

Recommendation: Implement calculation logic so that the dust values will not accumulate on the contract. One of the options can be a time-locked allowance to collect the accumulated dust value or to include the dust values as part of the rewards.

Found in: 0fd2faa

Status: Fixed (Revised commit: b2516c5)

Low

L01. Floating Pragma

Impact	Low
Likelihood	Low

The project uses floating pragmas ^0.8.0.

This may result in the contracts being deployed using the wrong pragma version, which is different from the one they were tested with. For example, they might be deployed using an outdated pragma version which may include bugs that affect the system negatively.

Paths:
./AMT.sol
./LIQUIDITYAMT.sol
./marketvault.sol
./Master.sol

Recommendation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment. Consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

Found in: e0fe204

Status: Fixed (Revised commit: b2516c5)

L02. Shadowing State Variable

Impact	Low
Likelihood	Low

The parameter *_name* of the AMT.constructor shadows an ERC20._name.

The parameter *_symbol* of the AMT.constructor shadows an ERC20._symbol.

The parameter *_name* of the LIQUIDITYAMT.constructor shadows an ERC20._name.

The parameter *_symbol* of the LIQUIDITYAMT.constructor shadows an ERC20._symbol.

Solidity allows for ambiguous naming of state variables when inheritance is used. Contract A with a variable x could inherit contract B, which also has a state variable x defined. This would result in two separate versions of x, one of them being accessed from contract A and the other one from contract B. In more complex contract systems, this condition could go unnoticed and subsequently lead to security issues.

Shadowing state variables can also occur within a single contract when there are multiple definitions on the contract and function level.

Paths:
./AMT.sol
./LIQUIDITYAMT.sol

Recommendation: Rename related variables/arguments.

Found in: e0fe204

Status: Fixed (Revised commit: 0fd2faa)

L03. Missing Zero Address Validation

Impact	Low
Likelihood	Low

Address parameters are used without checking against the possibility of 0x0.

This can lead to unwanted external calls to 0x0.

Paths:
./marketvault.sol : setMaster();
./Master.sol : liqLocker.constructor(), Master.constructor(), Master.setPayerWallet(), Master.mintMaster();

Recommendation: Implement zero address checks.

Found in: e0fe204

Status: Fixed (Revised commit: b2516c5)

L04. Invalid Hardcoded Value

Impact	Low
Likelihood	Low

In the *addLiquidityLocking()* and *addLiquidity()* functions, the *milisecsToValidate* variable is declared as ‘60000’ to represent 1 minute.

However, this is 1000 minutes, as EVM uses seconds, not milliseconds.

Path:
./Master.sol : addLiquidityLocking(), addLiquidity();

Recommendation: Change the value to ‘60’, correct the documentation or remove this variable.

When used in the equation *block.timestamp + milisecsToValidate* the *milisecsToValidate* has no effect as *block.timestamp* is sufficient as the deadline for the Uniswap.addLiquidity() function.

Found in: e0fe204

Status: Fixed (Revised commit: 0fd2faa)

Informational

I01. Variables That Should Be Declared Constant

Variables that do not change their value should be declared constant to save Gas.

Paths:
./AMT.sol : nameForDeploy, symbolForDeploy;
./LIQUIDITYAMT.sol : nameForDeploy, symbolForDeploy;
./Master.sol : addrRouter, liqRouter, posibleVariation, milisecsToValidate;

Recommendation: Declare variables as constants.

Found in: e0fe204

Status: Fixed (Revised commit: c674d1fc)

I02. Variables That Should Be Declared Immutable

The variables fee and sellRate are only declared in the constructor. These variables should be declared as immutable to save Gas.

Path:
./LIQUIDITYAMT.sol: fee, sellRate;

Recommendation: Declare the variables as immutable.

Found in: e0fe204

Status: Fixed (Revised commit: 0fd2faa)

I03. Unused Function Parameters

Parameters *_name* and *_symbol* of the constructor are never used.

Unused variables are allowed in Solidity and do not pose a direct security issue. It is best practice to avoid them as they can cause an increase in computations (and unnecessary Gas consumption) and decrease readability.

Paths:
./AMT.sol
./LIQUIDITYAMT.sol

Recommendation: Remove the unused parameters.

Found in: e0fe204

Status: Fixed (Revised commit: 0fd2faa)

I04. Unused Import

Address and *SafeMath* imports are never used in the LIQUIDITYAMT contract.

Unused imports should be removed from the contracts. Unused imports are allowed in Solidity and do not pose a direct security issue.

However, it is best practice to avoid them as they can decrease readability.

Path:
./LIQUIDITYAMT.sol

Recommendation: Remove the unused imports.

Found in: e0fe204

Status: Fixed (Revised commit: 0fd2faa)

I05. Use Of Hardcoded Values

In the AMT contract, total supply cap (100000000*(10**18)) is used in the mint function as hardcoded.

In the Master contract, on line 93 & 94, the amount to be approved is hardcoded.

In the Master contract, on line 197, lockingTime is hardcoded.

Using hardcoded values in the computations and comparisons is not best practice.

Paths:
./AMT.sol
./Master.sol

Recommendation: Convert this variable into constant.

Found in: e0fe204

Status: Fixed (Revised commit: 0fd2faa)

I06. Typos In The Documentation

In the *marketvault* contract, the word ‘withdrawl’ is spelled incorrectly several times. It should have been ‘withdrawal’.

In the *marketvault* contract, the word ‘Standar’ is spelled incorrectly. It should have been ‘Standard’.

In the *Master* contract, on line 133, the word ‘Liquity’ is spelled incorrectly. It should have been ‘Liquidity’.

In the *Master* contract, on line 79 and 80, the words ‘payed’ and ‘payd’ are spelled incorrectly. They should have been ‘paid’.

In the *Master* contract, on lines 176, 177, 223, 224, ‘to small’ is spelled incorrectly. It should have been ‘too small’.

Paths:
./marketvault.sol
./Master.sol

Recommendation: Fix the typos in the documentation.

Found in: e0fe204

Status: Fixed (Revised commit: b2516c5)

I07. Redundant Declaration

Boolean variables are already false by default. Therefore, there is no need to assign a value ‘false’ at the first declaration on line 63.

Path:
./Master.sol : liqLocked

Recommendation: Remove value assigning to save Gas.

Found in: e0fe204

Status: Fixed (Revised commit: b2516c5)

I08. Redundant Complexity

The condition in the require statement on lines 176, 177, 223, 224 has redundant complexity since it actually checks only if the amount is greater than 1.

Redundant complexity spends more Gas and decreases code readability.

Path:
./Master.sol: addLiquidityLocking(), addLiquidity();

Recommendation: Change the condition to amount > 1.

Found in: e0fe204

Status: Fixed (Revised commit: 0fd2faa)

I09. Style Guide Violation

Contract readability and code quality are influenced significantly by adherence to established style guidelines. In Solidity programming, there exist certain norms for code arrangement and ordering. These guidelines help to maintain a consistent structure across different contracts, libraries, or interfaces, making it easier for developers and auditors to understand and interact with the code.

The suggested order of elements within each contract, library, or interface is as follows:

1. Type declarations
2. State variables
3. Events
4. Modifiers
5. Functions

Functions should be ordered and grouped by their visibility as follows:

1. Constructor
2. Receive function (if exists)
3. Fallback function (if exists)
4. External functions
5. Public functions
6. Internal functions
7. Private functions

Within each grouping, view and pure functions should be placed at the end.

Furthermore, following the Solidity naming convention and adding NatSpec annotations for all functions are strongly recommended. These measures aid in the comprehension of code and enhance overall code quality.

The *marketvault* and *liqLocker* contracts are named with lowercase letters. Contracts and libraries should be named using the CapWords style.

Constant and immutable variables should be in uppercase.

Some functions violate line length standards. Please check 'Maximum Line Length' in the official Solidity guideline.

Paths:
./AMT.sol
./LIQUIDITYAMT.sol
./marketvault.sol
./Master.sol

Recommendation: Consistent adherence to the official Solidity style guide is recommended. This enhances readability and maintainability of the code, facilitating seamless interaction with the contracts. Providing comprehensive NatSpec annotations for functions and following Solidity's naming conventions further enrich the quality of the code. Follow the official Solidity guidelines.

Found in: e0fe204

Status: Reported (Revised commit: 0fd2faa) (Naming is not improved. Constant and immutable variables are not in uppercase)

I10. State Variables Default Visibility

Some variables' visibility is not specified in the project.

Specifying state variables visibility helps to catch incorrect assumptions about who can access the variable.

This makes the contract`s code quality and readability higher

Paths:
./AMT.sol : nameForDeploy, symbolForDeploy
./LiquidityAmt.sol: nameForDeploy, symbolForDeploy
./Market.sol: amt, btcb, usdt, addrBtcb, addrUsdt, fee, master
./liqLocker.sol: masterContract, btcb, liqToken
./Master.sol: liqLocked, amt, btcb, liqToken, externalLiqToken, addrRouter, liqRouter, liqFactory, addrBtcb, amountForApproval

Recommendation: Specify the intended visibility explicitly.

Found in: e0fe204

Status: Reported (*BurnVault.sol* visibilities and *maxAmt* variable in *Amt.sol* are not set.)(Revised commit: b2516c5)

I11. Unused Variable

masterSetControl variable is declared and never used in anywhere.

Unused variables are allowed in Solidity and do not pose a direct security issue. It is best practice to avoid them as they can cause an increase in computations (and unnecessary Gas consumption) and decrease readability.

Paths: ./Market.sol

Recommendation: Remove the unused variable.

Found in: 0fd2faa

Status: Fixed (Revised commit: b2516c5)

I12. Unused Import

The *Master.sol* contract is imported in the *BurnVault.sol* contract, but it is never used.

Unused imports are allowed in Solidity and do not pose a direct security issue. It is best practice to avoid them as they can cause an increase in computations (and unnecessary Gas consumption) and decrease readability.

Paths: ./BurnVault.sol

Recommendation: Remove the unused import.

Found in: 0fd2faa

Status: Fixed (Revised commit: b2516c5)

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with 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 report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

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.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

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

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	Critical	High	Medium
Medium Likelihood	High	Medium	Low
Low Likelihood	Medium	Low	Low

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/AutoMiningToken
Commit	e0fe2041cf5d09d711a49ebe8648dbdd16ea0b6f
Whitepaper	Link
Requirements	Link
Technical Requirements	-
Contracts Addresses	AMT.sol: https://bscscan.com/address/0x6ae0a238a6f51df8eee084b1756a54dd8a8e85d3 LIQUIDITYAMT.sol: https://bscscan.com/address/0x679bd76ca0b3f037131af9170d0462c9ffc9bc27 marketvault.sol: https://bscscan.com/address/0xaed5982fe57813312f883292b57c2bf924812cf0 Master.sol: https://bscscan.com/address/0x13e98112e1c67dbe684adf3aeb1c871f1fe6d1ac
Contracts	File: contracts/AMT.sol SHA3: c4f56cbf88f074d72299cf97465b9f54235b4046020bbecfe836448002d6a25e File: contracts/LIQUIDITYAMT.sol SHA3: 4b86dbccd06e2f6c836c77c11ad1e51b94fbef18b6d798c9d0022b9ff24d01d8 File: contracts/marketvault.sol SHA3: f617ba837df12552d64c599c27533a651083a5a6360784e3b96cf3dc7e861e81 File: contracts/Master.sol SHA3: 8e625087aae4ef4b7d7bb04b8e644b807b470ee3813e3612eb943acb9cc109a9

Second review scope

Repository	https://github.com/AutoMiningToken/amtEnviroment
Commit	0fd2faa756eb3bf197b3bbc1e7ad5dbccc51422f
Whitepaper	Link
Requirements	Link
Technical Requirements	-
Contracts	File: contracts/Amt.sol SHA3: 87d0054beb4bf8498681e8722dacfc06c514bf7dc28369b0ff066567aaa9d162

	File: contracts/BurnVault.sol SHA3: 1e6817fd9caf3c0714e4471bb8d8ea9bc392a176d6d28ef6d288f06d46ea0375 File: contracts/LiquidityAmt.sol SHA3: 3ffbe422c1890287d37de4e6331dadeeb812e62b488f00bcd0eacd5bce3468a9 File: contracts/Market.sol SHA3: 4cf3a0f8fda8ec1f7ad5fa60f1cd98b40b3c8b27c2bbd25ed56b53718c414735 File: contracts/Master.sol SHA3: f3ec3682eb2b9cad92ffcacf9326a94b55ded9ec79579d7ae7607fd4fa768edf

Third review scope

Repository	https://github.com/AutoMiningToken/amtEnviroment
Commit	b2516c5fbc7c6a1591d0ef57744f379476b26099
Whitepaper	Link
Requirements	Link
Technical Requirements	-
Contracts	File: contracts/Amt.sol SHA3: 9dbfac39f47100e40879ebbac20df902566cc2361b6d752661eb9778eb137101 File: contracts/BurnVault.sol SHA3: 94e35e9c0375ca88006fd40fc6daf236867374c525bc76b0157f20b8a7faf93b File: contracts/LiquidityAmt.sol SHA3: 1dc8b40da5f2adc102b2d6f01a144ea1ba017777e02cb89fb7e3d21c00b253bc File: contracts/Market.sol SHA3: a7999b77fa38622b7d41d81a496bf29f01fec3ec206b0fa70def705ed972f8bb File: contracts/Master.sol SHA3: 379226b21aa280efe85b91ae6899375244ed879667503a41a01b8ac52839fd49

Fourth review scope

Repository	https://github.com/AutoMiningToken/amtEnviroment
Commit	c674d1fcd1a3c6e1dd442d9407a5c789df8de9f3
Whitepaper	Link
Requirements	Link
Technical Requirements	-
Contracts	File: Production/contracts/Amt.sol SHA3: 9dbfac39f47100e40879ebbac20df902566cc2361b6d752661eb9778eb137101

	File: Production/contracts/BurnVault.sol SHA3: 673ed4b33e55b9aa62bfde717d2387aaae4dc9385e0bf2b2b8dbc9de0b327af0 File: Production/contracts/LiquidityAmt.sol SHA3: 1dc8b40da5f2adc102b2d6f01a144ea1ba017777e02cb89fb7e3d21c00b253bc File: Production/contracts/Market.sol SHA3: d1384046ad0767341532b36b2a4d87340ca2a18a7f2241eac6e774c28423ae32 File: Production/contracts/Master.sol SHA3: 9986a0dac27fea065b1ef4ef1f88bbc405f3837ffe29b474f5748a0024b4adac