ART ARMY PROTOCOL SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer:	Art Army Team (https://art.army)
Prepared on:	26/04/2021
Platform:	Binance Smart Chain
Language:	Solidity
Audit Type:	Standard

audit@etherauthority.io

Table of contents

Email: audit@EtherAuthority.io

THIS IS SECURITY AUDIT REPORT DOCUMENT AND WHICH MAY CONTAIN INFORMATION WHICH IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO PUBLIC AFTER ISSUES ARE RESOLVED.

Email: audit@EtherAuthority.io

Project files

Name	Smart Contract Code Review and Security Analysis Report for Art Army Protocol
Platform	Binance Smart Chain / Solidity
File 1	ArtArmyLiquidityLocked.sol
File 1 MD5 hash	E0551BB03E966898051C3F72A81AEC9C
File 2	ArtArmyStake.sol
File 2 MD5 hash	4FA8E42EC23B5EF0A0578B646A745A0E
File 3	ArtArmyToken.sol
File 3 MD5 hash	CCCAEF1E265620F0E4D3EFCE7F38B2C6

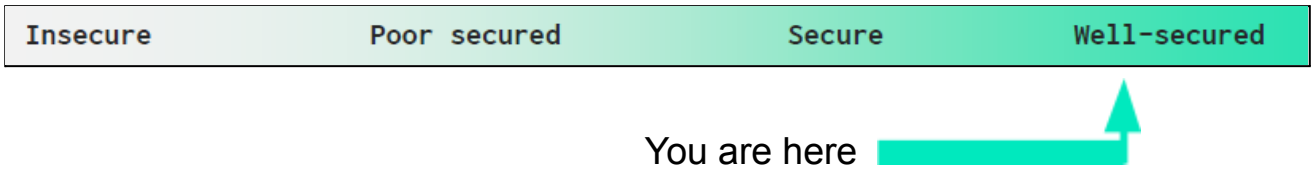
Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Moderated
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Passed
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Gas Optimization	Assert() misuse	Passed
	High consumption ‘for/while’ loop	Moderated
	High consumption ‘storage’ storage	Passed
	“Out of Gas” Attack	Passed
Business Risk	The maximum limit for mintage not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: PASSED

Executive Summary

According to the **extensive** audit assessment, Customer`s solidity smart contract is **well secured**.



We used various tools like SmartDec, Mythril, Slither and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.
All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Audit overview section. General overview is presented in AS-IS section and all found issues can be found in the Audit overview section.

We found 0 high, 0 medium and 1 low and some very low level issues.

Code Quality

Art Army protocol consists of 3 core smart contract files. These smart contracts also contain Libraries, Smart contract inherits and Interfaces. These are compact and well written contracts.

The libraries in the Art Army protocol are part of its logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned a specific address and its properties / methods can be reused many times by other contracts in the Art Army protocol.

The Art Army team has provided scenario and unit test scripts, which can be used to determine the integrity of the code in an automated way.

Overall, some code parts are **not** well commented. Commenting can provide rich documentation for functions, return variables and more. Ethereum Natural Language Specification Format (NatSpec) is recommended.

Documentation

We were given Art Army smart contracts in the form of solidity files. The hashes of those files are mentioned above in the table.

As mentioned above, most code parts are well commented (except some code parts). so anyone can quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol. It also provided a clear overview of the system components, including helpful details, like the lifetime of the background script.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure that are based on well known industry standard open source projects. And their core code blocks are written well.

Apart from libraries, Art Army smart contracts depend on an inter-connected set of smart contracts.

AS-IS overview

Art Army protocol is a decentralized gaming experience running on Binance Smart Chain, with other features like tokenization, liquidity locking, governance, etc. Following are the main components of core smart contracts.

ArtArmyLiquidityLocked.sol

(1) Imported contracts

(a) TokenTimelock.sol: This smart contract provides token locks. It is imported from open zeppelin and such external contracts are out of the scope of this audit, thus this smart contract is not audited in this SOP.

(2) Inherited contracts

(a) TokenTimelock :

(3) Functions

Sl.	Function	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue

ArtArmyStake.sol

(1) Interfaces
(a) IBEP20

(2) Imports
(a) SafeMath.sol
(b) Address.sol

(2) Usages
(a) using SafeBEP20 for IBEP20;
(b) using SafeMath for uint256;

(3) Events
(a) event StakeAdded(address indexed player);
(b) event StakeRemoved(address indexed player);
(c) event PlayerSet(address indexed player);

(4) Functions

Sl.	Function	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	gameName	read	Passed	No Issue
3	getStake	read	Passed	No Issue
4	getPlayerPoints	read	Passed	No Issue
5	getPlayerTime	read	Passed	No Issue
6	getPlayerAmount	read	Passed	No Issue
7	getPlayerCurrentPoints	read	Passed	No Issue
8	getPlayerExactTimePoints	read	Passed	No Issue
9	addStake	write	Passed	No Issue
10	removeStake	write	Passed	No Issue
11	_setPlayer	internal	Passed	No Issue

ArtArmyToken.sol

(1) Interfaces

(a) IBEP20

(2) Inherited contracts

(a) BEP20Token: Provides BEP20 token functions
(b) Context: Provide msg.sender and msg. value context
(c) IBEP20: Provides BEP20 interface
(d) Ownable: Provides ownership functions

(3) Usages

(a) using SafeMath for uint256

(4) events

(a) event Transfer(address indexed from, address indexed to, uint256 value);
(b) event Approval(address indexed owner, address indexed spender, uint256 value);
(c) event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
(d) event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed toDelegate);
(e) event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance);

(5) Functions

Sl	Function	Type	Observation	Conclusion	Score
1	getOwner	read	Passed	No Issue	Passed
2	decimals	read	Passed	No Issue	Passed
3	symbol	read	Passed	No Issue	Passed
4	name	read	Passed	No Issue	Passed
5	totalSupply	read	Passed	No Issue	Passed
6	balanceOf	read	Passed	No Issue	Passed
7	transfer	write	Passed	No Issue	Passed
8	allowance	read	Passed	No Issue	Passed
9	approve	write	Passed	No Issue	Passed
10	transferFrom	write	Passed	No Issue	Passed
11	increaseAllowance	write	Passed	No Issue	Passed
12	decreaseAllowance	write	Passed	No Issue	Passed
13	_transfer	internal	Passed	No Issue	Passed
14	_burn	internal	Passed	No Issue	Passed
15	_approve	internal	Passed	No Issue	Passed
16	_burnFrom	internal	Passed	No Issue	Passed
17	delegate	write	Passed	No Issue	Passed
18	delegateBySig	write	ECDSA Sig is used	User this feature carefully	Passed with consent
19	getCurrentVotes	read	Passed	No Issue	Passed
20	getPriorVotes	read	Infinite Loop Possibility	Keep Array Length Limited	Passed with consent
21	_delegate	internal	Passed	No Issue	Passed
22	_moveDelegates	internal	Passed	No Issue	Passed
23	_writeCheckpoint	internal	Passed	No Issue	Passed
24	safe32	internal	Passed	No Issue	Passed
25	add96	internal	Passed	No Issue	Passed
26	sub96	internal	Passed	No Issue	Passed
27	getChainId	internal	Passed	No Issue	Passed

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to tokens loss etc.
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 tokens lose
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.

Audit Findings

Critical

No critical severity vulnerabilities were found.

High

No high severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

(1) Infinite loops possibility:

```
    while (upper > lower) {
        uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
        Checkpoint memory cp = checkpoints[account][center];
        if (cp.fromBlock == blockNumber) {
            return cp.votes;
        } else if (cp.fromBlock < blockNumber) {
            lower = center;
        } else {
            upper = center - 1;
        }
    }
```

getPriorVotes function in ArtArmyToken.sol contract has the possibility that, if the *upper* value is too high than *lower*, then this loop will keep on going to hit the block's maximum limit .

Resolution: We got confirmation from the Art Army team that the array will be kept as limited length.

Very Low / Best Practice

(1) Ownership transfer function:

ArtArmyToken.sol smart contract has active ownership transfer. This will be troublesome if the ownership was sent to an incorrect address by human error.

```
    function _transferOwnership(address newOwner) internal {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}
```

so, it is a good practice to implement an acceptOwnership style to prevent it. Code flow similar to below:

```
    function transferOwnership(address payable _newOwner) external onlyOwner {
        newOwner = _newOwner;
    }

    //this flow is to prevent transferring ownership to wrong wallet by mistake
    function acceptOwnership() external {
        require(msg.sender == newOwner);
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
        newOwner = payable(0);
    }
}
```

Resolution: Art Army team acknowledged this, as this should be taken care of from admin side.

(2) Use the latest solidity version while contract deployment to prevent any compiler version level bugs.
Resolution: This issue is acknowledged.

(3) prefer using external visibility over public if that particular function is not used internally. It is considered as more efficient and saves some gas as well. https://ethereum.stackexchange.com/questions/19380/external-vs-public-best-practices/19391

(4) Unused code blocks: Ownership contract in ArtArmyToken.sol smart contract is not used anywhere. so if that is not required, then it's better to remove it to make the code clean.

Centralization

The ArtArmyStake contract has many functionality that will be handled from the server side such as selecting top 10 winners, etc.

Conclusion

We were given contract code. And we have used all possible tests based on given objects as files. The contracts are written so systematically, that we did not find any major issues. **So it is good to go for the production.**

Since possible test cases can be unlimited for such extensive smart contract protocol, so we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's 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 state of the reviewed contract, based on extensive audit procedure scope is "**Well Secured**".

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

EtherAuthority.io Disclaimer

EtherAuthority team has analyzed this smart contract 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).

Due to the fact that the total number of test cases are unlimited, so 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 also suggest to conduct a bug bounty program to confirm the high level of security of this smart contract.

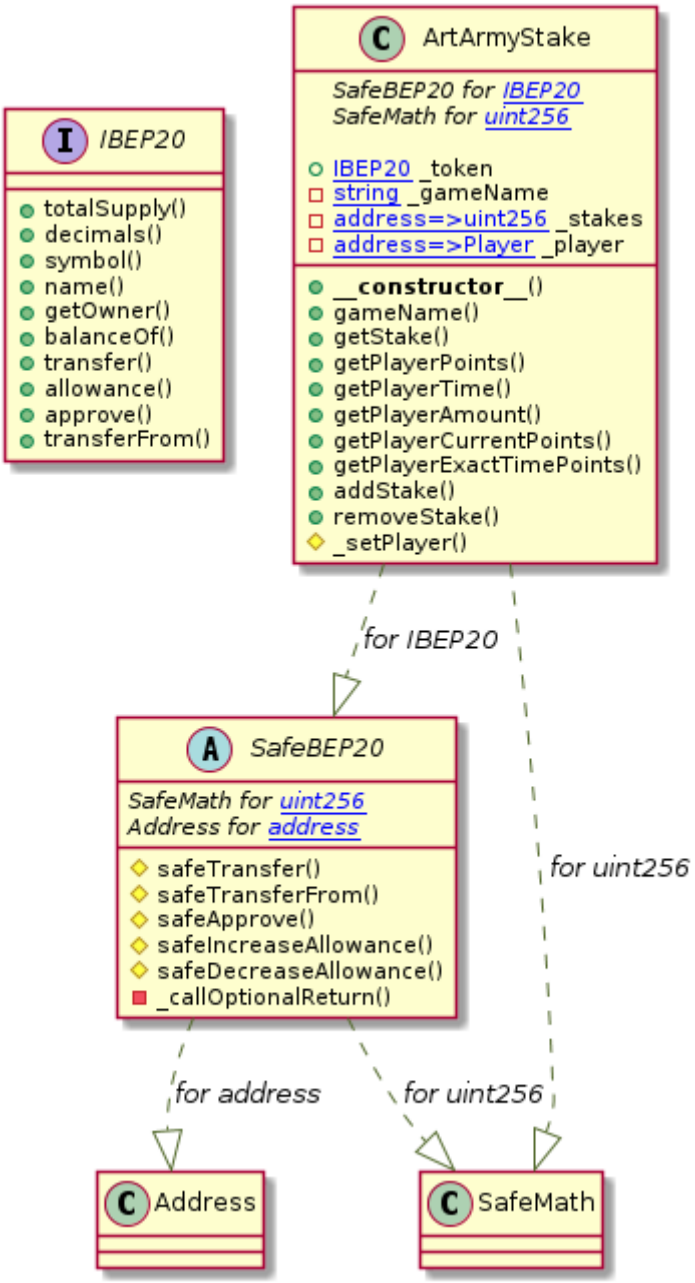
Technical Disclaimer

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

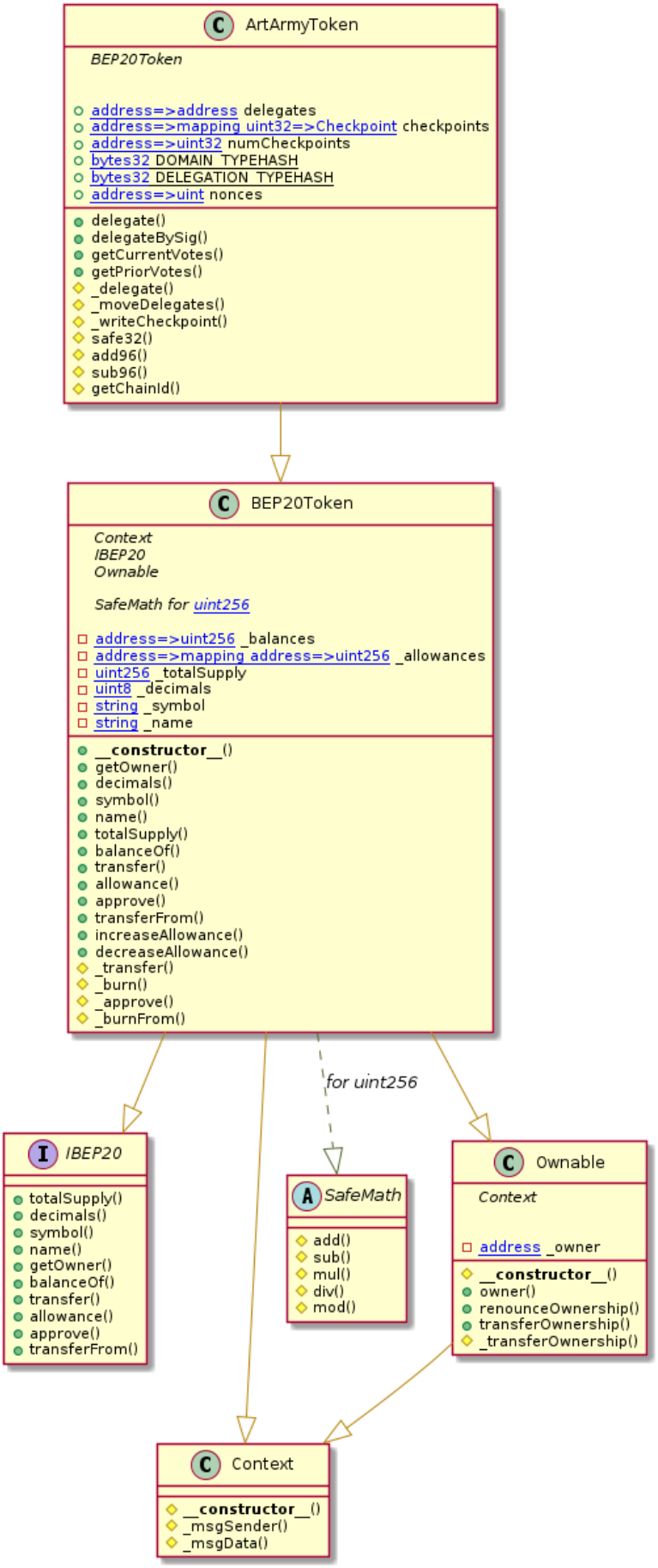
Appendix

Code Flow Diagram

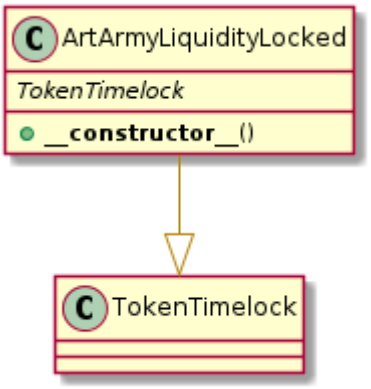
ArtArmyStake.sol



ArtArmyStake.sol



ArtArmyLiquidityLocked.sol



Slither log

root@server:/chetan# slither ArtArmyToken.sol
INFO:Detectors:
ArtArmyToken._writeCheckpoint(address,uint32,uint96,uint96)
(ArtArmyToken.sol#716-727) uses
a dangerous strict equality:
- nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock ==
blockNumber
(ArtArmyToken.sol#719)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strictequalities
INFO:Detectors:
BEP20Token.allowance(address,address).owner (ArtArmyToken.sol#424) shadows:
- Ownable.owner() (ArtArmyToken.sol#302-304) (function)
BEP20Token._approve(address,address,uint256).owner (ArtArmyToken.sol#550)
shadows:
- Ownable.owner() (ArtArmyToken.sol#302-304) (function)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
ArtArmyToken.delegateBySig(address,uint256,uint256,uint8,bytes32,bytes32)
(ArtArmyToken.sol#625-634) uses timestamp for comparisons
Dangerous comparisons:
- require(bool,string)(now <= expiry,Comp::delegateBySig: signature expired)
(ArtArmyToken.sol#632)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
ArtArmyToken.getChainId() (ArtArmyToken.sol#745-749) uses assembly
- INLINE ASM (ArtArmyToken.sol#747)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Redundant expression "this (ArtArmyToken.sol#119)" inContext
(ArtArmyToken.sol#109-122)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Detectors:
BEP20Token.constructor() (ArtArmyToken.sol#356-364) uses literals with too many digits:
- _totalSupply = 100000e18 (ArtArmyToken.sol#360)

Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
INFO:Detectors:
renounceOwnership() should be declared external:
- Ownable.renounceOwnership() (ArtArmyToken.sol#321-324)
transferOwnership(address) should be declared external:
- Ownable.renounceOwnership() (ArtArmyToken.sol#321-324)
transferOwnership(address) should be declared external:
- Ownable.transferOwnership(address) (ArtArmyToken.sol#330-332)
increaseAllowance(address,uint256) should be declared external:
- BEP20Token.increaseAllowance(address,uint256) (ArtArmyToken.sol#470-473)
decreaseAllowance(address,uint256) should be declared external:
- BEP20Token.decreaseAllowance(address,uint256) (ArtArmyToken.sol#489-492)
getPriorVotes(address,uint256) should be declared external:
- ArtArmyToken.getPriorVotes(address,uint256) (ArtArmyToken.sol#654-686)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-thatcould-be-declared-external
INFO:Slither:ArtArmyToken.sol analyzed (6 contracts with 72 detectors), 12 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration

root@server:/chetan# slither ArtArmyLiquidityLocked.sol
Compilation warnings/errors on ArtArmyLiquidityLocked.sol:
Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a
comment containing "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDXLicense-
Identifier: UNLICENSED" for non-open-source code. Please see https://spdx.org for more information.
--> ArtArmyLiquidityLocked.sol
Error: Source "@openzeppelin/contracts/token/ERC20/TokenTimelock.sol" not found: File not
found.

root@server:/chetan# slither ArtArmyStake.sol
Compilation warnings/errors on ArtArmyStake.sol:
Error: Source "@openzeppelin/contracts/math/SafeMath.sol" not found: File not found.
--> ArtArmyStake.sol:97:1:
|
97 | import "@openzeppelin/contracts/math/SafeMath.sol";
| ^^^
Error: Source "@openzeppelin/contracts/utils/Address.sol" not found: File not found.
--> ArtArmyStake.sol:98:1:
|
98 | import "@openzeppelin/contracts/utils/Address.sol";
| ^^^

Email: audit@EtherAuthority.io