CyStack

PUBLIC ACCESS

CYBERSECURITY AUDIT REPORT

Version v1.3

This document details the process and results of the smart contract audit performed independently by CyStack from 22/11/2021 to 14/01/2022.

Audited for

Alpha Omega Coin

Audited by

Vietnam CyStack Joint Stock Company

Contents

1 Introduction **4**
1.1 Audit Details 4
1.2 Audit Goals 5
1.3 Audit Methodology 6
1.4 Audit Scope 8

2 Executive Summary **10**

3 Detailed Results **14**

4 Conclusion **27**

5 Appendices **28**
Appendix A – Security Issue Status Definitions 28
Appendix B – Severity Explanation 29
Appendix C – Smart Contract Weakness Classification Registry (SWC Registry) . . . 30
Appendix D – Related Common Weakness Enumeration (CWE) 35

Disclaimer

Smart Contract Audit only provides findings and recommendations for an exact commitment of a smart contract codebase. The results, hence, are not guaranteed to be accurate outside of the commitment, or after any changes or modifications made to the codebase. The evaluation result does not guarantee the nonexistence of any further findings of security issues.

Time-limited engagements do not allow for a comprehensive evaluation of all security controls, so this audit does not give any warranties on finding all possible security issues of the given smart contract(s). CyStack prioritized the assessment to identify the weakest security controls an attacker would exploit. We recommend AlphaOmegaGlobal OÜ conducting similar assessments on an annual basis by internal, third-party assessors, or a public bug bounty program to ensure the security of smart contract(s).

This security audit should never be used as an investment advice.

Version History

Version	Date	Release notes
1.0	26/11/2021	The first report is sent to the client. All findings are in the open status.
1.1	25/12/2021	The report is sent to the client after retesting new codebases with fixed issues and additional features. Some findings are still not resolved, several new findings for additional features are addressed.
1.2	14/01/2022	The report is sent to the client after retesting new codebases, in which the issues remained from previous report were resolved. All the findings are resolved. AlphaOmegaGlobal OÜ allowed CyStack to publish the audit report publicly.
1.3	23/02/2022	AlphaOmegaGlobal OÜ requested to add information about the audited token's name, symbol, total supply, etc. in section 1.1. CyStack approved this request and confirmed that the additional information DOES NOT mislead any of the audit results.

Contact Information

Company	Representative	Position	Email address
AlphaOmegaGlob OÜ	Abraham Mankponsè Samuel	Chief Executive Officer	hello@aocfinance.com
CyStack	Vo Huyen Nhi	Sales Manager	nhivh@cystack.net
CyStack	Nguyen Tai Duc	Sales Executive	ducnt@cystack.net

Auditors

Fullname	Role	Email address
Nguyen Huu Trung	Head of Security	trungnh@cystack.net
Ha Minh Chau	Auditor	
Vu Hai Dang	Auditor	
Nguyen Van Huy	Auditor	
Nguyen Trung Huy Son	Auditor	
Nguyen Ba Anh Tuan	Auditor	

Introduction

From 22/11/2021 to 14/01/2022, AlphaOmegaGlobal OÜ engaged CyStack to evaluate the security posture of the Alpha Omega Coin of their contract system. Our findings and recommendations are detailed here in this initial report.

1.1 Audit Details

Audit Target

AOC stands for Alpha Omega Coin. In other words, call it GVC and in long God Virtual Coin.

Divinely inspired, it is intended to preserve humanity and in particular the true worshipers of God against the catastrophes of the sign 666, the sign of the antichrist, which is nothing else but a powerful global and global powerful economic system without which, to the glory of satan, one cannot buy, nor sell, nor do anything. Centralized system which to operate as a unique worldwide government. The one which is already gradually and fastly being put in place with ICTs and relating political decisions under the name of New World Order, the order of satan opposite to God's one.

The basic information of AOC is as follows:

Item	Description
Project Name	Alpha Omega Coin
Issuer	AlphaOmegaGlobal OÜ
Website	https://www.aocfinance.com/
Platform	Binance Smart Contract
Language	Solidity
Codebase (final version)	• **AOC Token:** https://testnet.bscscan.com/address/0x53cB59f3Ee1035daD6b63F88d9150EA70a1f2605#code • **AOC Proxy:** https://testnet.bscscan.com/address/0xBc79AE4eFE0259dCA865A0b75f6873d2EB0169A3#code
Commit	N/A
Audit method	Whitebox

AOC BEP-20 token displays the following characteristics:

- Name: Alpha Omega Coin;
- Symbol: AOC BEP20;
- Total supply: 1 000 000 000 000;
- Decimal: 18;
- Type: Utility token.

Audit Service Provider

CyStack is a leading security company in Vietnam with the goal of building the next generation of cybersecurity solutions to protect businesses against threats from the Internet. CyStack is a member of Vietnam Information Security Association (VNISA) and Vietnam Alliance for Cybersecurity Products Development.

CyStack's researchers are known as regular speakers at well-known cybersecurity conferences such as BlackHat USA, BlackHat Asia, Xcon, T2FI, etc. and are talented bug hunters who discovered critical vulnerabilities in global products and acknowledged by their vendors.

1.2 Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient and working according to its specifications. The audit activities can be grouped in the following three categories:

1. **Security:** Identifying security related issues within each contract and within the system of contracts.
2. **Sound Architecture:** Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. **Code Correctness and Quality:** A full review of the contract source code. The primary areas of focus include:
    - Correctness
    - Readability
    - Sections of code with high complexity
    - Improving scalability
    - Quantity and quality of test coverage

1.3 Audit Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology:

- **Likelihood** represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- **Impact** measures the technical loss and business damage of a successful attack;
- **Severity** demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: High, Medium and Low, i.e., H, M and L respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., Critical, Major, Medium, Minor and Informational (Info) as the table below:

Impact	Likelihood: *High*	Likelihood: *Medium*	Likelihood: *Low*
High	Critical	Major	Medium
Medium	Major	Medium	Minor
Low	Medium	Minor	Informational

Likelihood

CyStack firstly analyses the smart contract with open-source and also our own security assessment tools to identify basic bugs related to general smart contracts. These tools include Slither, securify, Mythril, Sūrya, Solgraph, Truffle, Geth, Ganache, Mist, Metamask, solhint, mythx, etc. Then, our security specialists will verify the tool results manually, make a description and decide the severity for each of them.

After that, we go through a checklist of possible issues that could not be detected with automatic tools, conduct test cases for each and indicate the severity level for the results. If no issues are found after manual analysis, the contract can be considered safe within the test case. Else, if any issues are found, we might further deploy contracts on our private testnet and run tests to confirm the findings. We would additionally build a PoC to demonstrate the possibility of exploitation, if required or necessary.

The standard checklist, which applies for every SCA, strictly follows the Smart Contract Weakness Classification Registry (SWC Registry). SWC Registry is an implementation of the weakness classification scheme proposed in The Ethereum Improvement Proposal project under the code EIP-1470. The checklist of testing according to SWC Registry is shown in Appendix A.

In general, the auditing process focuses on detecting and verifying the existence of the following issues:

- **Coding Specification Issues:** Focusing on identifying coding bugs related to general smart contract coding conventions and practices.
- **Design Defect Issues:** Reviewing the architecture design of the smart contract(s) and working on test cases, such as self-DoS attacks, incorrect inheritance implementations, etc.
- **Coding Security Issues:** Finding common security issues of the smart contract(s), for example integer overflows, insufficient verification of authenticity, improper use of cryptographic signature, etc.
- **Coding Design Issues:** Testing the code logic and error handlings in the smart contract code base, such as initializing contract variables, controlling the balance and flows of token transfers, verifying strong randomness, etc.
- **Coding Hidden Dangers:** Working on special issues, such as data privacy, data reliability, gas consumption optimization, special cases of authentication and owner permission, fallback functions, etc.

For better understanding of found issues' details and severity, each SWC ID is mapped to the most closely related Common Weakness Enumeration (CWE) ID. CWE is a category system for software weaknesses and vulnerabilities to help identify weaknesses surrounding software jargon. The list in Appendix B provides an overview on specific similar software bugs that occur in Smart Contract coding.

The final report will be sent to the smart contract issuer with an executive summary for overview and detailed results for acts of remediation.

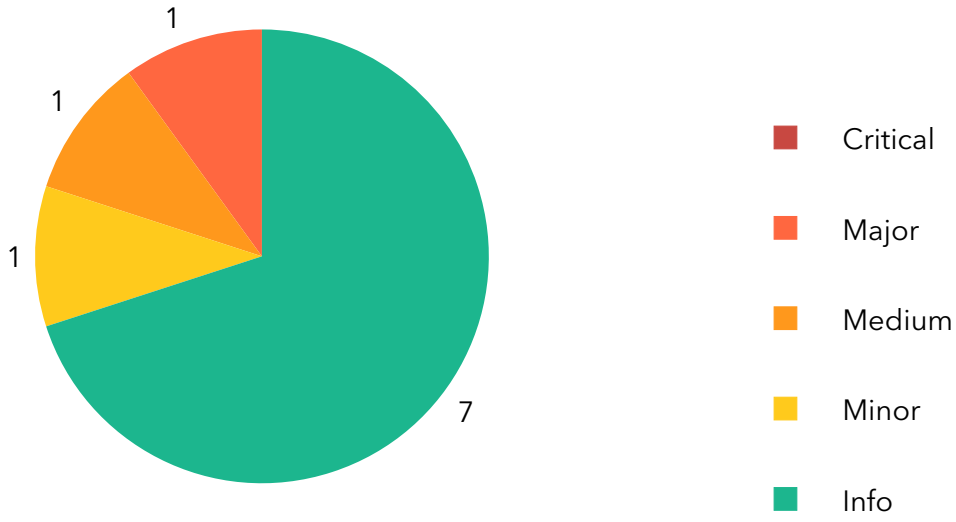
1.4 Audit Scope

Assessment	Target	Type
Original target		
White-box testing	AOC.sol	Solidity code file
Previous revision		
White-box testing	Contracts related to AOC Token	Solidity code file
White-box testing	Contracts related to AOC Proxy	Solidity code file
Last revision		
White-box testing	AOC_BEP.sol	Solidity code file
White-box testing	DateTime.sol	Solidity code file
White-box testing	StorageSlotUpgradeable.sol	Solidity code file
White-box testing	ContextUpgradeable.sol	Solidity code file
White-box testing	AddressUpgradeable.sol	Solidity code file
White-box testing	IERC20MetadataUpgradeable.sol	Solidity code file
White-box testing	IERC20Upgradeable.sol	Solidity code file
White-box testing	PausableUpgradeable.sol	Solidity code file
White-box testing	UUPSUpgradeable.sol	Solidity code file
White-box testing	Initializable.sol	Solidity code file
White-box testing	IBeaconUpgradeable.sol	Solidity code file
White-box testing	ERC1967UpgradeUpgradeable.sol	Solidity code file
White-box testing	OwnableUpgradeable.sol	Solidity code file
White-box testing	import.sol	Solidity code file
White-box testing	ERC1967Proxy.sol	Solidity code file
White-box testing	TransparentUpgradeableProxy.sol	Solidity code file
White-box testing	ProxyAdmin.sol	Solidity code file

White-box testing	Proxy.sol	Solidity code file
White-box testing	ERC1967Upgrade.sol	Solidity code file
White-box testing	IBeacon.sol	Solidity code file
White-box testing	Address.sol	Solidity code file
White-box testing	StorageSlot.sol	Solidity code file
White-box testing	Ownable.sol	Solidity code file
White-box testing	Context.sol	Solidity code file
White-box testing	UUPSUpgradeable.sol	Solidity code file
White-box testing	Proxiable.sol	Solidity code file

Executive Summary

Security issues by severity



Security issues by SWC

Floating Pragma (SWC-103)	1	
Unprotected Ether Withdrawal (SWC-105)	1	
State Variable Default Visibility (SWC-108)	2	
Incorrect Constructor Name (SWC-118)	2	
Requirement Violation (SWC-123)	1	
Typographical Error (SWC-129)	2	
Code With No Effects (SWC-135)	1	

Security issues by CWE

Improper Access Control (CWE-284)	1
Use of Incorrect Operator (CWE-480)	2
Improper Following of Specification by Caller (CWE-573)	1
Improper Control of a Resource Through its Lifetime (CWE-664)	1
Improper Initialization (CWE-665)	2
Improper Adherence to Coding Standards (CWE-710)	2
Irrelevant Code (CWE-1164)	1

Table of security issues

ID	Status	Vulnerability	Severity
#aoc-001	Resolved	Floating pragma	INFO
#aoc-002	Resolved	Improper state of visibility for the variable *owner*	INFO
#aoc-003	Resolved	Missing error messages	MINOR
#aoc-004	Resolved	Incorrect implementation for the function *withdrawBNBFromContract*	INFO
#aoc-005	Resolved	High gas cost due to improper initialization of variables	INFO
#aoc-006	Resolved	Defect in coding design for the contract constructor	INFO
#aoc-007	Resolved	Misnamed functions in contract codebases	INFO
#aoc-008	Resolved	Possibility of BNB withdrawals to *address(0)*	MAJOR

#aoc-009	Resolved	Timestamps do not match with terms in Regressive Anti-Manipulating Strategy (RAMS)	INFO
#aoc-010	Resolved	Large Transaction Authorisation Feature (LTAF) and RAMS do not work properly for balance with a small value	MEDIUM

Recommendations

Based on the results of this smart contract audit, CyStack has the following high-level key recommendations:

Key recommendations	
Issues	CyStack conducted the third SCA for AOC after AOC team had committed new codebases with mitigations for remaining issues from the second report. All the findings are resolved.
Recommendations	CyStack recommends AlphaOmegaGlobal OÜ to evaluate the audit results with several different security audit third-parties for the most accurate conclusion.
References	• https://consensys.github.io/smart-contract-best-practices/known_attacks • https://consensys.github.io/smart-contract-best-practices/recommendations/ • https://medium.com/@knownsec404team/ethereum-smart-contract-audit-checklist-ba9d1159b901

Detailed Results

1. Floating pragma

Issue ID	#aoc-001
Category	SWC-103 - Floating Pragma
Description	Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.
Severity	INFO
Location(s)	AOC_BEP.sol:3, import.sol:2
Status	Resolved
Reference	CWE-664 - Improper Control of a Resource Through its Lifetime
Remediation	Lock the pragma version and also consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

Description

Taking AOC_BEP.sol as an example for this issue, the codeline where floating pragma is used:

```
…
3  pragma solidity ^0.8.0;
…
```

The code can be revised as following:

```
…
3  pragma solidity 0.8.0;
…
```

or:

```
…
3  pragma solidity 0.8.7;
…
```

2. Improper state of visibility for the variable *owner*

Issue ID	#aoc-002
Category	SWC-108 - State Variable Default Visibility
Description	The variable *owner* is initialized in the contract *Ownable*. This variable stores the address of contract owner. However, the variable should be initialized with the specifier *private* instead of *public*, in order to prevent public accesses from other contracts and optimize the gas cost when deploying the contract and conducting transactions.
Severity	INFO
Location(s)	AOC.sol:119
Status	Resolved
Reference	CWE-710 - Improper Adherence to Coding Standards
Remediation	Initialize *owner* as a private variable and add a getter function for *owner*.

Description

The codelines where the issue occurs:

```
…
118  contract Ownable is Context {
119    address public owner;
120
121    /**
122     * @dev The Ownable constructor sets the original `owner` of the contract to
     ↪  the sender
123     * account.
124     */
125    constructor() {
126      owner = msg.sender;
127    }
…
```

The code can be revised as following:

```
…
118 contract Ownable is Context {
119   address private owner;
120   function owner() public view returns (address) {
121    return _owner;
122 }
123
124   /**
125    * @dev The Ownable constructor sets the original `owner` of the contract to
    ↪  the sender
126    * account.
127    */
128   constructor() {
129     owner = msg.sender;
130   }
…
```

3. Missing error messages

Issue ID	#aoc-003
Category	SWC-123 - Requirement Violation
Description	The require can be used to check for conditions and throw an exception if the condition is not met. It is better to provide a string message containing details about the error that will be passed back to the caller.
Severity	MINOR
Location(s)	AOC.sol:133, 555, 567
Status	Resolved
Reference	CWE-573 - Improper Following of Specification by Caller
Remediation	Add error messages to every *require()*.

Description

The codelines where the issue occurs in BEP-20:

```
…
133     require(msg.sender == owner);
…
555     require(amount <= address(this).balance);
…
567     require(tokenContract.balanceOf(address(this)) > _amount);
…
```

The code can be revised as following:

```
…
133     require(msg.sender == owner, "Ownable: caller is not the owner");
…
555     require(amount <= address(this).balance, "withdrawBNBFromContract:
        ↪ withdraw amount exceeds BNB balance");
…
567     require(tokenContract.balanceOf(address(this)) > _amount, "withdrawToken:
        ↪ withdraw amount exceeds token balance");
…
```

4. Incorrect implementation for the function *withdrawBNBFromContract*

Issue ID	#aoc-004
Category	SWC-135 Code With No Effects
Description	In the contract for AOC, the function *withdrawBNBFromContract* is implemented for withdrawals BNB from the contract to the address of *owner*. However, no "receive" function is implemented for transaction of BNB to the *owner* address, by which means that the *owner* cannot receive BNB.
Severity	INFO
Location(s)	AOC.sol:554-559
Status	Resolved
Reference	CWE-1164 - Irrelevant Code
Remediation	Add the additional implementations, e.g. *receive() external payable{} or fallback() external payable{}*, so that the *owner* can receive BNB to their address.

Description

The codelines where the issue occurs:

```
…
554     function withdrawBNBFromContract(uint256 amount) public onlyOwner {
555         require(amount <= address(this).balance);
556         address payable _owner = payable(msg.sender);
557         _owner.transfer(amount);
558         emit EthFromContractTransferred(amount);
559     }
…
```

5. High gas cost due to improper initialization of variables

Issue ID	#aoc-005
Category	SWC-108 - State Variable Default Visibility Severity
Description	One-time initilized variables should be specified as *constant* instead of private.
Severity	INFO
Location(s)	AOC.sol:246-248
Status	Resolved
Reference	CWE-710 - Improper Adherence to Coding Standards
Remediation	Use the specifier *constant* for these variables instead of *private*.

Description

The codelines where the issue occurs:

```
…
246     uint8 private _decimal = 18;
247     string private _name = "Alpha Omega Coin";
248     string private _symbol = "AOC";
…
```

6. Defect in coding design for the contract constructor

Issue ID	#aoc-006
Category	SWC-118 - Incorrect Constructor Name
Description	Before constructor execution, the variable *_totalSupply* has already been initialized with the value 1000 * 10 ** 9 * 10**18. However, the function *_mint* is executed in the constructor, which makes the value of *_totalSupply* doubled. This leads to incorrect amount of total supply.
Severity	INFO
Location(s)	AOC.sol:245,284,516
Status	Resolved
Reference	CWE-665 - Improper Initialization
Remediation	Do not initialize *_totalSupply* with a concrete value, fix it with *uint256 private _totalSupply;*, then change the value of _totalSupply using *_mint*.

Description

The codelines where the issue occurs:

```
…
245     uint256 private _totalSupply = 1000 * 10**9 * 10**18;
…
283     constructor () {
284         _mint(_msgSender(), _totalSupply);
285     }
…
516     function _mint(address account, uint256 amount) internal virtual {
517         require(account != address(0), "BEP20: mint to the zero address");
518
519         _beforeTokenTransfer(address(0), account, amount);
…
521         _totalSupply += amount;
522         _balances[account] += amount;
523         emit Transfer(address(0), account, amount);
524     }
525
```

The code can be revised as following:

```
…
245     uint256 private _totalSupply;
…
283     constructor () {
284         _mint(_msgSender(), 1000 * 10**9 * 10**18);
285     }
…
516     function _mint(address account, uint256 amount) internal virtual {
517         require(account != address(0), "BEP20: mint to the zero address");
518
519         _beforeTokenTransfer(address(0), account, amount);
…
521         _totalSupply += amount;
522         _balances[account] += amount;
523         emit Transfer(address(0), account, amount);
524     }
525
```

7. Misnamed functions in contract codebases

Issue ID	#aoc-007
Category	SWC-129 - Typographical Error
Description	The event *EthFromContractTransferred* should be renamed as *BnbFromContractTransferred* instead for clearance.
Severity	INFO
Location(s)	AOC.sol:257, 558
Status	Resolved
Reference	CWE-480 - Use of Incorrect Operator
Remediation	Rename *EthFromContractTransferred* as *BnbFromContractTransferred*.

8. Possibility of BNB withdrawals to *address(0)*

Issue ID	#aoc-008
Category	SWC-105 - Unprotected Ether Withdrawal
Description	Function *withdrawBNBFromContract* does not check the address of *recipient*. If *address(0)* is taken as the address of *recipient*, there will be a loss of BNB.
Severity	MAJOR
Location(s)	AOC_BEP.sol:359-363
Status	Resolved
Reference	CWE-284 - Improper Access Control
Remediation	Add a *require* statement to validate the address of *recipient*.

Description

The codelines where the issue occurs:

```
...
359     function withdrawBNBFromContract(address payable recipient, uint256
        ↪ amount) external onlyOwner payable {
360         require(amount <= address(this).balance, "withdrawBNBFromContract:
            ↪ withdraw amount exceeds BNB balance");
361         recipient.transfer(amount);
362         emit BNBFromContractTransferred(amount);
363     }
...
```

The code can be revised as following:

```
...
359     function withdrawBNBFromContract(address payable recipient, uint256
        ↪ amount) external onlyOwner payable {
360         require(recipient != address(0), "Address cant be zero address");
361         require(amount <= address(this).balance, "withdrawBNBFromContract:
            ↪ withdraw amount exceeds BNB balance");
362         recipient.transfer(amount);
363         emit BNBFromContractTransferred(amount);
364     }
...
```

9. Timestamps do not match with terms in Regressive Anti-Manipulating Strategy (RAMS)

Issue ID	#aoc-009
Category	SWC-118 - Incorrect Constructor Name
Description	The timestamp in the codebase do not match what stated in the RAMS.
Severity	INFO
Location(s)	AOC_BEP.sol:113-126
Status	Resolved
Reference	CWE-665 - Improper Initialization
Remediation	Redefine the levels with correct timestamps corresponding to the RAMS description.

Description

From the RAMS document:

1. Per month, and from January 1, 2022 to January 1, 2024, AOC BEP-20 holders cannot surpass a transaction of 20% of the total balance they have on the wallet during first transaction done on the month.
2. Per month, and from January 2, 2024 to January 1, 2026, AOC BEP-20 holders cannot surpass a transaction of 15% of the total balance they have on the wallet during first transaction done on the month.
3. Per month, and from January 2, 2026 to January 1, 2028, AOC BEP-20 holders cannot surpass a transaction of 10% of the total balance they have on the wallet during first transaction done on the month.
4. Per month, and from January 2, 2028 till the end of this world, AOC BEP-20 holders cannot surpass a transaction of 5% of the total balance they have on the wallet during first transaction done on the month.

The codelines where the issue occurs:

```
…
113     function initialize() public initializer {
114         _mint(_msgSender(), (1000 * 10**9 * 10**18)); //mint the initial total
            ↪ supply
115         ltafPercentage = 50;
116
117         addLevels(1, 1609459200, 1640995140, 20); //01/01/2021 - 31/12/2021
118         addLevels(2, 1640995200, 1672531140, 15); //01/01/2022 - 31/12/2022
119         addLevels(3, 1672531200, 1704067140, 10); //01/01/2022 - 31/12/2022
120         addLevels(4, 1704067200, 0, 5);           //01/01/2024 -
121
122         // initializing
123         __Pausable_init_unchained();
124         __Ownable_init_unchained();
125         __Context_init_unchained();
126     }
…
```

The timestamps passed to *addLevels* define the dates in the additional code comments.

10. Large Transaction Authorisation Feature (LTAF) and RAMS do not work properly for balance with a small value

Issue ID	#aoc-010
Category	SWC-129 - Typographical Error
Description	The validation require statements for LTAF and RAMS in the function *_transfer* do not properly handle the value of user balances. If a balance holds a value smaller than 0.5 * 10**18, the result of amount / 10**18 equals to 0. This leads to a bypass against the LTAF and RAMS restrictions.
Severity	MEDIUM
Location(s)	AOC_BEP.sol:393-406
Status	Resolved
Reference	CWE-480 - Use of Incorrect Operator
Remediation	Remove the division by 10**18 in the math expressions.

Description

The codelines where the issue occurs:

```
394         if(includedInLTAF[sender] || !excludedFromRAMS[sender]) {
395             // convert current timestamp to uint256
396             (uint256 year, uint256 month, uint256 day) =
                ↪ DateTimeLibrary.timestampToDate(block.timestamp);
397             if(day == 1 || year != userInfo[sender].year || month !=
                ↪ userInfo[sender].month || userInfo[sender].level == 0)
                ↪ updateUserInfo(sender, year, month);
398
399             if(includedInLTAF[sender]) {
400                 // validate amount
401                 require((amount / 10**18 ) <= (((userInfo[sender].balance /
                    ↪ 10**18) * ltafPercentage) / 10**2), "BEP20: Amount is
                    ↪ higher than LTAF percentage");
402             } else if(!excludedFromRAMS[sender]) {
403                 // validate amount
```

```
404                 if(userInfo[sender].level > 0) require((amount / 10**18 ) <=
                    ↪  (((userInfo[sender].balance / 10**18) *
                    ↪  levels[userInfo[sender].level].percentage) / 10**2),
                    ↪  "BEP20: Amount is higher");
405             }
406         }
```

The code can be revised as following:

```
394         if(includedInLTAF[sender] || !excludedFromRAMS[sender]) {
395             // convert current timestamp to uint256
396             (uint256 year, uint256 month, uint256 day) =
                ↪  DateTimeLibrary.timestampToDate(block.timestamp);
397             if(day == 1 || year != userInfo[sender].year || month !=
                ↪  userInfo[sender].month || userInfo[sender].level == 0)
                ↪  updateUserInfo(sender, year, month);
398
399             if(includedInLTAF[sender]) {
400                 // validate amount
401                 require(amount <= ((userInfo[sender].balance * ltafPercentage)
                    ↪  / 10**2), "BEP20: Amount is higher than LTAF percentage");
402             } else if(!excludedFromRAMS[sender]) {
403                 // validate amount
404                 if(userInfo[sender].level > 0) require(amount <=
                    ↪  ((userInfo[sender].balance *
                    ↪  levels[userInfo[sender].level].percentage) / 10**2),
                    ↪  "BEP20: Amount is higher");
405             }
406         }
```

Conclusion

CyStack had conducted a security audit for Alpha Omega Coin Token. Total ten issues were found, including a major, a medium, a minor and seven informational issues. Right after receiving the first audit report, the Alpha Omega Coin issuer immediately take action on addressing the issue based on their severity. CyStack confirmed that all found issues were resolved. Overall, Alpha Omega Coin has included the best practices for smart contract development and has passed our security assessment for smart contracts.

To improve the quality for this report, and for CyStack's Smart Contract Audit report in general, we greatly appreciate any constructive feedback or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

Appendices

Appendix A – Security Issue Status Definitions

Status	Definition
Open	The issue has been reported and currently being review by the smart contract developers/issuer.
Unresolved	The issue is acknowledged and planned to be addressed in future. At the time of the corresponding report version, the issue has not been fixed.
Resolved	The issue is acknowledged and has been fully fixed by the smart contract developers/issuer.
Rejected	The issue is considered to have no security implications or to make only little security impacts, so it is not planned to be addressed and won't be fixed.

Appendix B – Severity Explanation

Severity	Definition
CRITICAL	Issues, considered as critical, are straightforwardly exploitable bugs and security vulnerabilities. It is advised to immediately resolve these issues in order to prevent major problems or a full failure during contract system operation.
MAJOR	Major issues are bugs and vulnerabilities, which cannot be exploited directly without certain conditions. It is advised to patch the codebase of the smart contract as soon as possible, since these issues, with a high degree of probability, can cause certain problems for operation of the smart contract or severe security impacts on the system in some way.
MEDIUM	In terms of medium issues, bugs and vulnerabilities exist but cannot be exploited without extra steps such as social engineering. It is advised to form a plan of action and patch after high-priority issues have been resolved.
MINOR	Minor issues are generally objective in nature but do not represent actual bugs or security problems. It is advised to address these issues, unless there is a clear reason not to.
INFO	Issues, regarded as informational (info), possibly relate to “guides for the best practices” or “readability”. Generally, these issues are not actual bugs or vulnerabilities. It is recommended to address these issues, if it make effective and secure improvements to the smart contract codebase.

Appendix C – Smart Contract Weakness Classification Registry (SWC Registry)

ID	Name	Description
	Coding Specification Issues	
SWC-100	Function Default Visibility	It is recommended to make a conscious decision on which visibility type (*external*, *public*, *internal* or *private*) is appropriate for a function. By default, functions without concrete specifiers are *public*.
SWC-102	Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler to avoid publicly disclosed bugs and issues in outdated versions.
SWC-103	Floating Pragma	It is recommended to lock the pragma to ensure that contracts do not accidentally get deployed using.
SWC-108	State Variable Default Visibility	Variables can be specified as being *public*, *internal* or *private*. Explicitly define visibility for all state variables.
SWC-111	Use of Deprecated Solidity Functions	Solidity provides alternatives to the deprecated constructions, the use of which might reduce code quality. Most of them are aliases, thus replacing old constructions will not break current behavior.
SWC-118	Incorrect Constructor Name	It is therefore recommended to upgrade the contract to a recent version of the Solidity compiler and change to the new constructor declaration (the keyword *constructor*).
	Design Defect Issues	
SWC-113	DoS with Failed Call	External calls can fail accidentally or deliberately, which can cause a DoS condition in the contract. It is better to isolate each external call into its own transaction and implement the contract logic to handle failed calls.

SWC-119	Shadowing State Variables	Review storage variable layouts for your contract systems carefully and remove any ambiguities. Always check for compiler warnings as they can flag the issue within a single contract.
SWC-125	Incorrect Inheritance Order	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order (from more /general/ to more /specific/).
SWC-128	DoS With Block Gas Limit	Modifying an array of unknown size, that increases in size over time, can lead to such a Denial of Service condition. Actions that require looping across the entire data structure should be avoided.
	Coding Security Issues	
SWC-101	Integer Overflow and Underflow	It is recommended to use safe math libraries for arithmetic operations throughout the smart contract system to avoid integer overflows and underflows.
SWC-107	Reentrancy	Make sure all internal state changes are performed before the call is executed or use a reentrancy lock.
SWC-112	Delegatecall to Untrusted Callee	Use *delegatecall* with caution and make sure to never call into untrusted contracts. If the target address is derived from user input ensure to check it against a whitelist of trusted contracts.
SWC-117	Signature Malleability	A signature should never be included into a signed message hash to check if previously messages have been processed by the contract.
SWC-121	Missing Protection against Signature Replay Attacks	In order to protect against signature replay attacks, store every message hash that has been processed by the smart contract, include the address of the contract that processes the message and never generate the message hash including the signature.
SWC-122	Lack of Proper Signature Verification	It is not recommended to use alternate verification schemes that do not require proper signature verification through *ecrecover()*.

SWC-130	Right-To-Left-Override control character (U+202E)	The character *U+202E* should not appear in the source code of a smart contract.
	Coding Design Issues	
SWC-104	Unchecked Call Return Value	If you choose to use low-level call methods (e.g. *call()*), make sure to handle the possibility that the call fails by checking the return value.
SWC-105	Unprotected Ether Withdrawal	Implement controls so withdrawals can only be triggered by authorized parties or according to the specs of the smart contract system.
SWC-106	Unprotected SELFDESTRUCT Instruction	Consider removing the self-destruct functionality. If absolutely required, it is recommended to implement a multisig scheme so that multiple parties must approve the self-destruct action.
SWC-110	Assert Violation	Consider whether the condition checked in the *assert()* is actually an invariant. If not, replace the *assert()* statement with a *require()* statement.
SWC-116	Block values as a proxy for time	Developers should write smart contracts with the notion that block values are not precise, and the use of them can lead to unexpected effects. Alternatively, they may make use oracles.
SWC-120	Weak Sources of Randomness from Chain Attributes	To avoid weak sources of randomness, use commitment scheme, e.g. RANDAO, external sources of randomness via oracles, e.g. Oraclize, or Bitcoin block hashes.
SWC-123	Requirement Violation	If the required logical condition is too strong, it should be weakened to allow all valid external inputs. Otherwise, make sure no invalid inputs are provided.
SWC-124	Write to Arbitrary Storage Location	As a general advice, given that all data structures share the same storage (address) space, one should make sure that writes to one data structure cannot inadvertently overwrite entries of another data structure.

SWC-132	Unexpected Ether balance	Avoid strict equality checks for the Ether balance in a contract.
SWC-133	Hash Collisions With Multiple Variable Length Arguments	When using *abi.encodePacked()*, it's crucial to ensure that a matching signature cannot be achieved using different parameters. Alternatively, you can simply use *abi.encode()* instead. It is also recommended to use replay protection.
	Coding Hidden Dangers	
SWC-109	Uninitialized Storage Pointer	Uninitialized local storage variables can point to unexpected storage locations in the contract. If a local variable is sufficient, mark it with *memory*, else *storage* upon declaration. As of compiler version 0.5.0 and higher this issue has been systematically resolved.
SWC-114	Transaction Order Dependence	A possible way to remedy for race conditions in submission of information in exchange for a reward is called a commit reveal hash scheme. The best fix for the ERC20 race condition is to add a field to the inputs of approve which is the expected current value and to have approve revert or add a safe approve function.
SWC-115	Authorization through tx.origin	*tx.origin* should not be used for authorization. Use *msg.sender* instead.
SWC-126	Insufficient Gas Griefing	Insufficient gas griefing attacks can be performed on contracts which accept data and use it in a sub-call on another contract. To avoid them, only allow trusted users to relay transactions and require that the forwarder provides enough gas.
SWC-127	Arbitrary Jump with Function Type Variable	The use of assembly should be minimal. A developer should not allow a user to assign arbitrary values to function type variables.

SWC-129	Typographical Error	The weakness can be avoided by performing pre-condition checks on any math operation or using a vetted library for arithmetic calculations such as SafeMath developed by OpenZeppelin.
SWC-131	Presence of unused variables	Remove all unused variables from the code base.
SWC-134	Message call with hardcoded gas amount	Avoid the use of *transfer()* and *send()* and do not otherwise specify a fixed amount of gas when performing calls. Use *.call.value(...)("")* instead.
SWC-135	Code With No Effects	It's important to carefully ensure that your contract works as intended. Write unit tests to verify correct behaviour of the code.
SWC-136	Unencrypted Private Data On-Chain	Any private data should either be stored off-chain, or carefully encrypted.

Appendix D – Related Common Weakness Enumeration (CWE)

The SWC Registry loosely aligned to the terminologies and structure used in the CWE while overlaying a wide range of weakness variants that are specific to smart contracts.

CWE IDs *, to which SWC Registry is related, are listed in the following table:

CWE ID	Name	Related SWC IDs
CWE-284	**Improper Access Control**	SWC-105, SWC-106
CWE-294	Authentication Bypass by Capture-replay	SWC-133
CWE-664	**Improper Control of a Resource Through its Lifetime**	SWC-103
CWE-123	Write-what-where Condition	SWC-124
CWE-400	Uncontrolled Resource Consumption	SWC-128
CWE-451	User Interface (UI) Misrepresentation of Critical Information	SWC-130
CWE-665	Improper Initialization	SWC-118, SWC-134
CWE-767	Access to Critical Private Variable via Public Method	SWC-136
CWE-824	Access of Uninitialized Pointer	SWC-109
CWE-829	Inclusion of Functionality from Untrusted Control Sphere	SWC-112, SWC-116
CWE-682	**Incorrect Calculation**	SWC-101
CWE-691	**Insufficient Control Flow Management**	SWC-126
CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization (“Race Condition”)	SWC-114
CWE-480	Use of Incorrect Operator	SWC-129
CWE-667	Improper Locking	SWC-132
CWE-670	Always-Incorrect Control Flow Implementation	SWC-110
CWE-696	Incorrect Behavior Order	SWC-125
CWE-841	Improper Enforcement of Behavioral Workflow	SWC-107
CWE-693	**Protection Mechanism Failure**	

CWE-330	Use of Insufficiently Random Values	SWC-120
CWE-345	Insufficient Verification of Data Authenticity	SWC-122
CWE-347	Improper Verification of Cryptographic Signature	SWC-117, SWC-121
CWE-703	**Improper Check or Handling of Exceptional Conditions**	SWC-113
CWE-252	Unchecked Return Value	SWC-104
CWE-710	**Improper Adherence to Coding Standards**	SWC-100, SWC-108, SWC-119
CWE-477	Use of Obsolete Function	SWC-111, SWC-115
CWE-573	Improper Following of Specification by Caller	SWC-123
CWE-695	Use of Low-Level Functionality	SWC-127
CWE-1164	Irrelevant Code	SWC-131, SWC-135
CWE-937	**Using Components with Known Vulnerabilities**	SWC-102

** CWE IDs, which are presented in bold, are the greatest parent nodes of those nodes following it.*

All IDs in the CWE list above are relevant to the view "Research Concepts" (CWE-1000), except for CWE-937, which is relevant to the "Weaknesses in OWASP Top Ten (2013)" (CWE-928).