CyStack

PUBLIC ACCESS

# CYBERSECURITY AUDIT REPORT

## Version v1.2

*This document details the process and results of the smart contract audit performed independently by CyStack from 03/11/2021 to 10/11/2021.*

*Audited for*

**RICE Token**

*Audited by*

**Vietnam CyStack Joint Stock Company**

# Contents



## Independent Audit Report Disclaimer

This document is an independent smart contract audit report, which is the result of CyStack's independent security assessment for a smart contract. This conducted audit strictly follows terms and conditions, publicly stated by the smart contract issuer.

## Disclaimer

Smart Contract Audit only provides findings and recommendations for an exact commitment of a smart contract codebase. The results, hence, is 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 recommends Rice Decentralized Finance ecosystem 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 | 07/11/2021 | The first report was sent to the contract issuer.<br>All findings were in open status. |
| 1.1 | 09/11/2021 | The second report was made after the contract issuer had their responses to every found issue. |
| 1.2 | 10/11/2021 | The contract issuer allowed CyStack to publish the audit report publicly. |

## Auditors

| Fullname | Role | Email address | Phone number |
|---|---|---|---|
| Nguyen Huu Trung | Head of Security | trungnh@cystack.net | (+84) 974 914 322 |
| Ha Minh Chau | Auditor | | |
| Vu Hai Dang | Auditor | | |
| Nguyen Van Huy | Auditor | | |
| Nguyen Trung Huy Son | Auditor | | |
| Nguyen Ba Anh Tuan | Auditor | | |

# Introduction

From 03/11/2021 to 10/11/2021, CyStack independently evaluated the security posture of the smart contract RICE Token from the Rice Decentralized Finance ecosystem. Our findings and recommendations are detailed here in this initial report.

**NOTE:** The report will be continually updated to correctly reflect the mitigation and remediation state of each finding.

## 1.1 Audit Details

### Audit Target

RICE Token (RICE) is a utility token that can only be used within the Rice Wallet application and the Rice Decentralized Finance ecosystem.

Rice Wallet is a decentralized financial application that allows users to store and manage their digital assets with absolute control (private key or seed phrase). Besides, Rice Wallet will help make it easier for investors to access the decentralized financial (Defi) market. With the carefully selected decentralized applications (Dapps) and customized UX/UI such as Swap (DEX), Staking, Investing, Pooling, etc you can explore the entire Defi market from one place.

The basic information of RICE is as follows:

| Item | Description |
|---|---|
| Project Name | RICE Token |
| Issuer | Rice Decentralized Finance ecosystem |
| Website | https://ricewallet.io/ |
| Platform | Ethereum Smart Contract |
| Language | Solidity |
| Codebase | https://etherscan.io/address/0xBCD515D6C5de70D3A31D999A7FA6a299657De294#code |
| Commit | N/A |
| Audit method | Whitebox |

### 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 |
| | | *High* | *Medium* | *Low* |

**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.

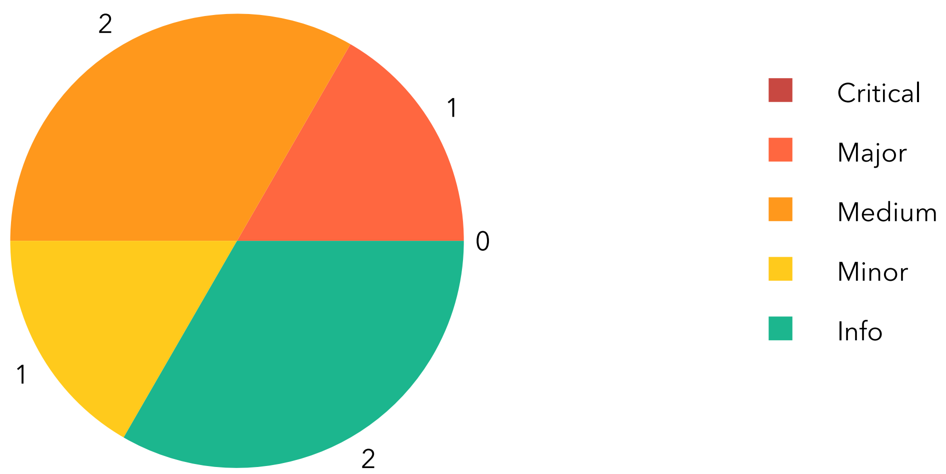
## 1.4 Audit Scope

| Assessment | Target | Type |
|---|---|---|
| White-box testing | RICEToken.sol | Solidity code file |

# Executive Summary

## Security issues by severity

Legend




## Security issues by SWC

| | |
|---|---|
| Integer Overflow and Underflow (SWC-101) | 1 |
| State Variable Default Visibility (SWC-108) | 1 |
| Requirement Violation (SWC-123) | 1 |
| Incorrect Inheritance Order (SWC-125) | 1 |
| DoS With Block Gas Limit (SWC-128) | 1 |
| Unexpected Ether Balance (SWC-132) | 1 |

## Security issues by CWE

| | |
|---|---|
| Uncontrolled Resource Consumption (CWE-400) | 1 |
| Improper Following of Specification by Caller (CWE-573) | 1 |
| Improper Locking (CWE-667) | 1 |
| Incorrect Calculation (CWE-682) | 1 |
| Incorrect Behavior Order (CWE-696) | 1 |
| Improper Adherence to Coding Standards (CWE-710) | 1 |

## Table of security issues

| ID | Status | Vulnerability | Severity |
|---|---|---|---|
| #rice-001 | Unresolved | Logical issue of *transferWithLock()* | MAJOR |
| #rice-002 | Unresolved | Unbound release date | MEDIUM |
| #rice-003 | Unresolved | Division before multiplication | MEDIUM |
| #rice-004 | Unresolved | Passable *pause()* on transfers of tokens | MINOR |
| #rice-005 | Unresolved | Missing Mutability Specifiers | INFO |
| #rice-006 | Unresolved | Missing Error Messages | INFO |

## Recommendations

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

| Key recommendations | |
|---|---|
| Issues | CyStack has conducted SCA for Rice Token and detected some significant issues that require immediate acts of mitigation, listed below:<br>• Several functions can be executed without checking essential conditions.<br>• Variables are initialized and *require()* is called without considering of optimization.<br>• Math operations are incorrectly performed. |
| Recommendations | • Carefully check for conditions with *require()* before every function execution.<br>• Optimize the use of variables and *require()*.<br>• Perform math operation correctly, this could be improve by using publicly provided math libraries. |
| References | • https://consensys.github.io/smart-contract-best-practices/known_attacks<br>• https://consensys.github.io/smart-contract-best-practices/recommendations/<br>• https://media.consensys.net/when-to-use-revert-assert-and-require-in-solidity-61fb2c0e5a57 |

# Detailed Results

## 1. Logical issue of *transferWithLock()*

| | |
|---|---|
| **Issue ID** | #rice-001 |
| **Category** | SWC-132 - Unexpected Ether Balance |
| **Description** | By the function *transferWithLock()*, any members of the FoundingTeam, PrivateSales, or the owner are allowed to send tokens without verifying whether they are locked or unlocked. Moreover, members with the mentioned roles can send tokens to themselves. |
| **Severity** | MAJOR |
| **Location(s)** | RICEToken.sol:974~984 |
| **Status** | Unresolved |
| **Reference** | CWE-667 - Improper Locking |
| **Remediation** | Revise the design and implementation of the function *transferWithLock()*. Improve centralized privileges or roles in the protocol via a decentralized mechanism, e.g., via multisignature wallet. |

### Description

The codelines where the issue occurs:

```
…
974     function transferWithLock(address _receiver, uint256 _amount, uint256 _releaseDate) public returns (bool success) {
975         require(msg.sender == FoundingTeam || msg.sender == PrivateSales || msg.sender == owner());
976         ERC20._transfer(msg.sender,_receiver,_amount);
977
978         if (lockList[_receiver].length==0) lockedAddressList.push(_receiver);
979
980         LockTime memory item = LockTime({amount:_amount, releaseDate:_releaseDate});
981         lockList[_receiver].push(item);
982
983         return true;
984     }
…
```

The amount of locked tokens stored in *lockList[_receiver]*, with a sufficiently high release date, is used by the *getLockedAmount()* function to compute the *getAvailableBalance()* function, which checks if a transfer can take place. If a privileged member sends too much token to himself/herself, the locked amount of that member will be falsely increased, so *getAvailableBalance()* will be reverted due to a subtraction overflow. In consequence, *transfer()* and *transferFrom()* will become unworkable with the member address as a sender. This problem can also happen if two or more privileged members make token transfers between themselves in a certain way, using this function.

The code can be revised as following:

```
…
    function transferWithLock(address _receiver, uint256 _amount, uint256 _releaseDate) public returns (bool success) {
        require(msg.sender == FoundingTeam || msg.sender == PrivateSales || msg.sender == owner());
        require(getAvailableBalance(_msgSender()) >= _amount, "transferWithLock: transfer amount exceeds available balance ");
        ERC20._transfer(msg.sender,_receiver,_amount);

        if (lockList[_receiver].length==0) lockedAddressList.push(_receiver);

        LockTime memory item = LockTime({amount:_amount, releaseDate:_releaseDate});
        lockList[_receiver].push(item);

        return true;
    }
…
```

## 2. Unbound release date

| | |
|---|---|
| **Issue ID** | #rice-002 |
| **Category** | SWC-128 - DoS With Block Gas Limit |
| **Description** | The function *transferWithLock()* allows members of the FoundingTeam, PrivateSales and the owner to transfer and block a quantity of tokens for a certain period of time. However, since *_releaseDate* has no upper bound, tokens could be blocked for an extremely long time. |
| **Severity** | MEDIUM |
| **Location(s)** | RICEToken.sol:974~984 |
| **Status** | Unresolved |
| **Reference** | CWE-400 - Uncontrolled Resource Consumption |
| **Remediation** | Set an upper bound to *_releaseDate* in the function *transferWithLock()* to avoid extreme dates. |

### Description

The codelines where the issue occurs:

```
…
974     function transferWithLock(address _receiver, uint256 _amount, uint256
        ↪ _releaseDate) public returns (bool success) {
975         require(msg.sender == FoundingTeam || msg.sender == PrivateSales || msg.sender
            ↪ == owner());
976         ERC20._transfer(msg.sender,_receiver,_amount);
977
978         if (lockList[_receiver].length==0) lockedAddressList.push(_receiver);
979
980         LockTime memory item = LockTime({amount:_amount, releaseDate:_releaseDate});
981         lockList[_receiver].push(item);
982
983         return true;
984     }
…
```

## 3. Division before multiplication

| | |
|---|---|
| **Issue ID** | #rice-003 |
| **Category** | SWC-101 - Integer Overflow and Underflow |
| **Description** | In the file RICEToken.sol, division is performed before multiplication. Multiplication should be performed before division in order to avoid loss of precision. |
| **Severity** | MEDIUM |
| **Location(s)** | RICEToken.sol:1055 |
| **Status** | Unresolved |
| **Reference** | CWE-682 - Incorrect Calculation |
| **Remediation** | Reorder the two operations: perform multiplication before division. |

**Description**

The codeline where the issue occurs:

```
…
1053    function lockInvestor(uint256 investorId) public onlyOwner {
1054        for(uint y = 3; y <= 10; y++) {
1055        transferWithLock(investorsList[investorId].wallet,
            ↪  (investorsList[investorId].amount / 8) * 10 ** uint256(decimals()),
            ↪  PrivateSalesMap[y]);
1056        }
1057    }
…
```

The operations should be reorder, so that multiplication is applied before division in the calculation:

```
…
1053    function lockInvestor(uint256 investorId) public onlyOwner {
1054        for(uint y = 3; y <= 10; y++) {
1055        transferWithLock(investorsList[investorId].wallet,
            ↪  (investorsList[investorId].amount) * 10 ** uint256(decimals()) / 8,
            ↪  PrivateSalesMap[y]);
1056        }
1057    }
…
```

# 4. Passable *pause()* on transfers of tokens

| | |
|---|---|
| **Issue ID** | #rice-004 |
| **Category** | SWC-125 - Incorrect Inheritance Order |
| **Description** | The functions *transfer()* and *transferFrom()* of the RICEToken contract invoke the functions transfer() and transferFrom() from the ERC20 module. This bypasses the Pausable module from the ERC20Pausable inheritance and allows the transfer of these tokens to successfully pass even when *paused()* returns *true*. |
| **Severity** | MINOR |
| **Location(s)** | RICEToken.sol:963, 971 |
| **Status** | Unresolved |
| **Reference** | CWE-696 - Incorrect Behavior Order |
| **Remediation** | Invoke *super.transfer()* and *super.transferFrom()* in the functions *transfer()* and *transferFrom()* instead of those from the ERC20 module. |

### Description

The codelines where the issue occurs:

```
...
960     function transfer(address _receiver, uint256 _amount) public returns (bool
        ↪ success) {
961         require(_receiver != address(0));
962         require(_amount <= getAvailableBalance(msg.sender));
963         return ERC20.transfer(_receiver, _amount);
964     }
965
966     function transferFrom(address _from, address _receiver, uint256 _amount) public
        ↪ returns (bool) {
967         require(_from != address(0));
968         require(_receiver != address(0));
969         require(_amount <= allowance(_from, msg.sender));
970         require(_amount <= getAvailableBalance(_from));
971         return ERC20.transferFrom(_from, _receiver, _amount);
972     }
...
```

If the mentioned functions were designed to be pausable, it is recommended to follows the above remediation. Otherwise, it should be clearly commented in the codebase.

## 5. Missing Mutability Specifiers

| | |
|---|---|
| **Issue ID** | #rice-005 |
| **Category** | SWC-108 - State Variable Default Visibility |
| **Description** | The linked variables are assigned to only once, either during their contract-level declaration or during the constructor's execution. This unnecessarily increases the gas cost of utilizing the variables. |
| **Severity** | INFO |
| **Location(s)** | RICEToken.sol:891, 892, 893, 894, 912, 913, 914 |
| **Status** | Unresolved |
| **Reference** | CWE-710 - Improper Adherence to Coding Standards |
| **Remediation** | Optimize the gas cost involved in utilizing the variables with the use of the additional keyword *immutable* for lines 891-894 (Solidity up to and above v0.6.5.) and the use of keyword *constant* for lines 912-914 instead of *private*. |

### Description

The codelines where the issue occurs:

```
…
891     address FoundingTeam = 0x12B8665E7b4684178a54122e121B83CC41d9d9C3;
892     address UserAcquisition = 0xdf7E62218B2f889a35a5510e65f9CD4288CB6D6E;
893     address PublicSales = 0x876443e20778Daa70BFd2552e815A674D0aA7BF8;
894     address PrivateSales = 0x20b803C1d5C9408Bdc5D76648A6F23EB519CD2bD;
…
912     uint8 private _d = 18;
913     uint256 private totalTokens = 1000000000 * 10 ** uint256(_d);
914     uint256 private initialSupply = 600000000 * 10 ** uint256(_d);
…
```

Our recommendation:

```
…
891     address immutable FoundingTeam = 0x12B8665E7b4684178a54122e121B83CC41d9d9C3;
892     address immutable UserAcquisition = 0xdf7E62218B2f889a35a5510e65f9CD4288CB6D6E;
893     address immutable PublicSales = 0x876443e20778Daa70BFd2552e815A674D0aA7BF8;
894     address immutable PrivateSales = 0x20b803C1d5C9408Bdc5D76648A6F23EB519CD2bD;
…
912     uint8 constant _d = 18;
913     uint256 constant totalTokens = 1000000000 * 10 ** uint256(_d);
914     uint256 constant initialSupply = 600000000 * 10 ** uint256(_d);
…
```

# 6. Missing Error Mesages

| | |
|---|---|
| **Issue ID** | #rice-006 |
| **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** | INFO |
| **Location(s)** | RICEToken.sol:962, 969, 970, 975 |
| **Status** | Unresolved |
| **Reference** | CWE-573 - Improper Following of Specification by Caller |
| **Remediation** | Provide a string message containing details about the error. |

### Description

The codelines where the issue occurs:

```
…
962        require(_amount <= getAvailableBalance(msg.sender));
…
969        require(_amount <= allowance(_from, msg.sender));
970        require(_amount <= getAvailableBalance(_from));
…
975        require(msg.sender == FoundingTeam || msg.sender == PrivateSales || msg.sender
           ↪  == owner());
…
```

Our recommendation:

```
…
962        require(_amount <= getAvailableBalance(msg.sender), ``RICEToken: insufficient
           ↪  funds to transfer'');
…
969        require(_amount <= allowance(_from, msg.sender), ``RICEToken: allowance to
           ↪  low'');
970        require(_amount <= getAvailableBalance(_from), ``RICEToken: insufficient funds
           ↪  to transfer'');
…
975        require(msg.sender == FoundingTeam || msg.sender == PrivateSales || msg.sender
           ↪  == owner(), ``RICEToken: You're not allowed to use this function'');
…
```

# 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.<br>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.<br>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.<br>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.<br>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 C – 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).*