

PUBLIC ACCESS

CYBERSECURITY AUDIT REPORT

Version v1.2

This document details the process and results of the smart contract audit performed by CyStack from 30/03/2022 to 14/04/2022.

Audited for

NFT5

Audited by

Vietnam CyStack Joint Stock Company

© 2022 CyStack. All rights reserved.

Portions of this document and the templates used in its production are the property of CyStack and cannot be copied (in full or in part) without CyStack's permission.

While precautions have been taken in the preparation of this document, CyStack the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of CyStack's services does not guarantee the security of a system, or that computer intrusions will not occur.

Contents

1	Introduction	4
1.1	Audit Details	4
1.2	Audit Goals	7
1.3	Audit Methodology	7
1.4	Audit Scope	9
2	Executive Summary	10
3	Detailed Results	12
4	Conclusion	15
5	Appendices	16
	Appendix A – Security Issue Status Definitions	16
	Appendix B – Severity Explanation	17
	Appendix C – Smart Contract Weakness Classification Registry (SWC Registry)	18
	Appendix D – Related Common Weakness Enumeration (CWE)	23

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 NFT5 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	05/04/2022	Security assessments for AssetHolder.sol had been done. No issues were found.
1.1	14/04/2022	Security assessments for all smart contracts of NFT5 had completed. One of the two issues was immediately resolved, the other remained unresolved.
1.2	15/04/2022	All issues were resolved. NFT5 allowed CyStack to publish the audit report publicly.

Contact Information

Company	Representative	Position	Email address
NFT5	Nguyen Hoang Trieu	Product Owner	trieunguyen@nft5.io
CyStack	Vo Huyen Nhi	Sales Manager	nhivh@cystack.net
CyStack	Nguyen Ngoc Anh	Sales Executive	anhntn@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 30/03/2022 to 14/04/2022, NFT5 engaged CyStack to evaluate the security posture of the NFT5 platform. Our findings and recommendations are detailed here in this initial report.

1.1 Audit Details

Audit Target

NFT5 is the next-generation NFT platform. NFTs issued on NFT5 will represent part of the ownership and earnings from the intellectual property product and provide users seamless experience from contacting, trading, buying NFTs in the most secure, blazing fast and convenient way.

NFT5 is built and developed by the team of Remitano, under the NFT5.

The basic information of this project is detailed in the following table:

Item	Description
Project Name	NFT5 platform (GitHub: remitano/nft5-core)
Issuer	NFT5
Website	https://nft5.io/
Platform	Binance Smart Contract
Language	Solidity
Codebase	https://github.com/remitano/nft5-core/tree/409747c7758d4a2b52e39a700a1f57fcd377dc57/contracts
Commit	409747c7758d4a2b52e39a700a1f57fcd377dc57
Audit method	Whitebox

ERC-1155 is the token standard selected by the NFT5 development team as the token standard operating throughout the NFT5 system. ERC-1155 is a standard that helps smart contracts manage multiple token types. A smart contract that applies the ERC-1155 standard can handle many different types of tokens such as fungible, non-fungible or semi-fungible.

NFT5 business model is illustrated in the following figure:



Main features of NFT5 that will be introduced:

- **Minting NFT (Completed)**: This is a function that helps content creators publish their NFTs through NFT5 as a bridge;
- **Marketplace (Completed)**: This is a marketplace that circulates all unique crypto assets (NFTs) of NFT5 users, with quality smart contracts, world-class security transaction;
- **Staking NFT (In Progress)**: Investors who agree to lock a part of their NFTs will be rewarded based on certain criteria;
- **Auction (In Progress)**: Auctioned NFTs will increase competition among collectors, creating excitement for the NFT trading market at NFT5.

In the scope of this security assessment, CyStack evaluated the security posture of only two features: **Minting NFT** and **Marketplace**. Four contracts were implemented in NFT5 platform for these features:

- AssetHolder.sol;
- FeeCollector.sol;
- MarketPlace.sol;
- MintNFT.sol.

In the first part of the audit, CyStack focused on analyzing **AssetHolder.sol** and finding possible security issues in this contract. The contract AssetHolder.sol functions as a storage of BUSD from users on NFT5 platform. Also, by this contract, users are approved to make transactions of NFTs on NFT5 Marketplace.

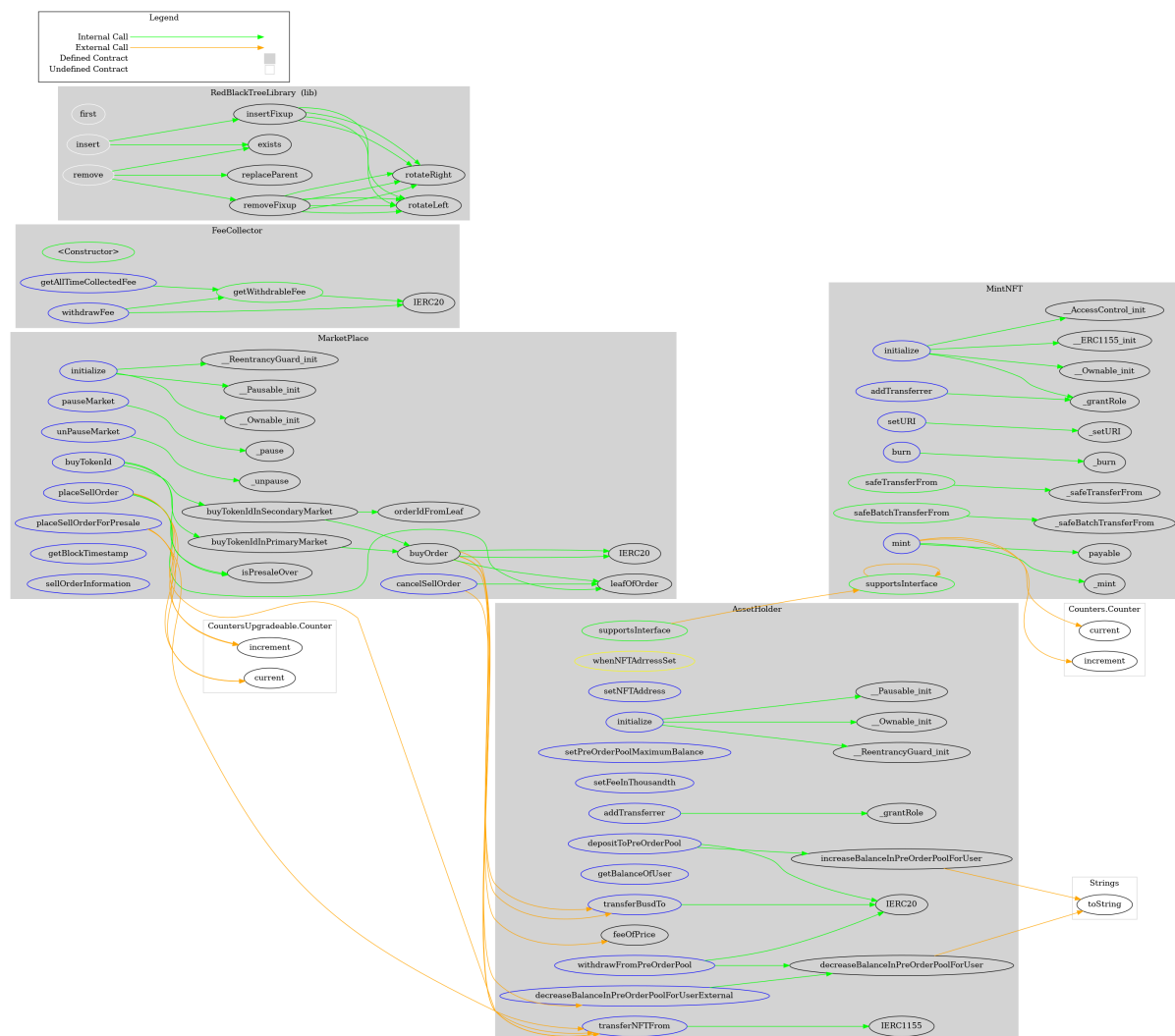
In the second part of the audit, CyStack conducted security assessments on the three contracts FeeCollector.sol, MarketPlace.sol and MintNFT.sol.

The contract MarketPlace.sol allows users to sell and purchase NFT items. Sellers can create a sell order with arbitrary amount and unit price. When buyers make their purchases, buy orders are prioritised from low to high. Buyers can also choose any accounts from the pool (AssetHolder) to purchase NFT items. Additionally, the owner of the contract MarketPlace.sol can create private sales for certain NFTs in a defined duration and can pause/unpause MarketPlace.sol.

The contract MintNFT.sol performs functions related NFTs, including minting, burning and transferring NFT items on NFT5 ecosystem. This contract is designed with two privileges: ADMIN (owner) and TRANSFERER. Only ADMIN can mint and burn existing NFTs.

The contract FeeCollector.sol functions as a marketplace fee collector, by which means, every transaction fee on NFT5's marketplace (MarketPlace.sol) is transferred to this contract. The collected fees are all stored in the contract. Only the owner of FeeCollector.sol can interact with this contract. The owner can view information about the contract balance and total collected fee. Also, the owner can directly transfer all BUSD in the contract balance to his/her address.

The function calls in of NFT5 project is illustrated in the following graph:



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	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 C.

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 D 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.

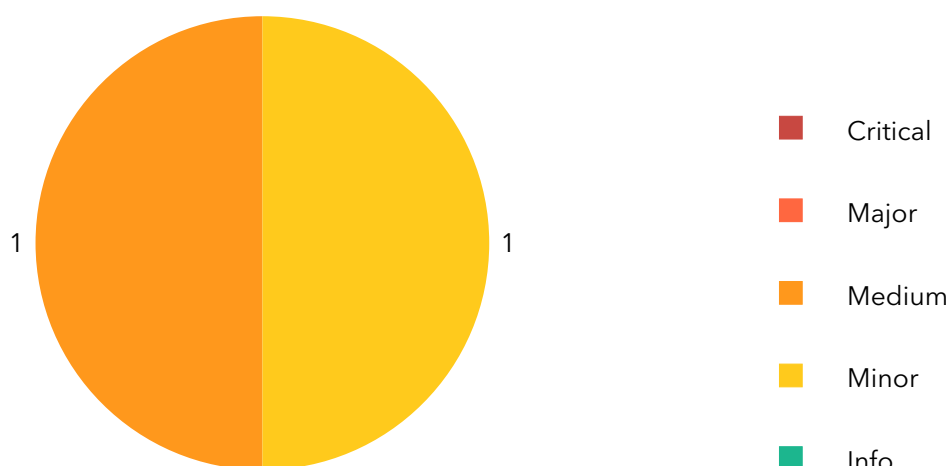
1.4 Audit Scope

Assessment	Target	Type
Original target		
White-box testing	AssetHolder.sol	Solidity code file
White-box testing	FeeCollector.sol	Solidity code file
White-box testing	MarketPlace.sol	Solidity code file
White-box testing	MintNFT.sol	Solidity code file
Previous revision		
White-box testing	AssetHolder.sol	Solidity code file
White-box testing	FeeCollector.sol	Solidity code file
White-box testing	MarketPlace.sol	Solidity code file
White-box testing	MintNFT.sol	Solidity code file
Last revision		
White-box testing	AssetHolder.sol	Solidity code file
White-box testing	FeeCollector.sol	Solidity code file
White-box testing	MarketPlace.sol	Solidity code file
White-box testing	MintNFT.sol	Solidity code file

Executive Summary

Security issues by severity

Legend



Security issues by SWC

Integer Overflow and Underflow (SWC-101)

2 ■ ■

Security issues by CWE

Incorrect Calculation (CWE-682)

2 ■ ■

Table of security issues

ID	Status	Vulnerability	Severity
#nft5-001	Resolved	Integer overflow via order.price	MEDIUM
#nft5-002	Resolved	Integer overflow via order.id	MINOR

Recommendations

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

Key recommendations	
Issues	<p>CyStack had completed the smart contract audit for NFT5 platform. In the first part of the audit, no issues were found for AssetHolder.sol.</p> <p>In the second part, CyStack conducted security assessments on the three contracts FeeCollector.sol, MarketPlace.sol and MintNFT.sol. Total two issues were found and all then were resolved rapidly by NFT5.</p>
Recommendations	<p>CyStack recommends NFT5 to evaluate the audit results with several different security audit third-parties for the most accurate conclusion.</p>
References	<ul style="list-style-type: none">• 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

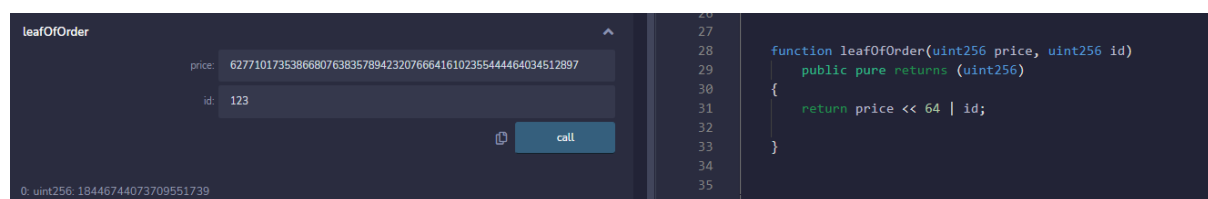
Detailed Results

1. Uncontrolled value of *order.price* leads to malfunctioned *orderIdFromLeaf*

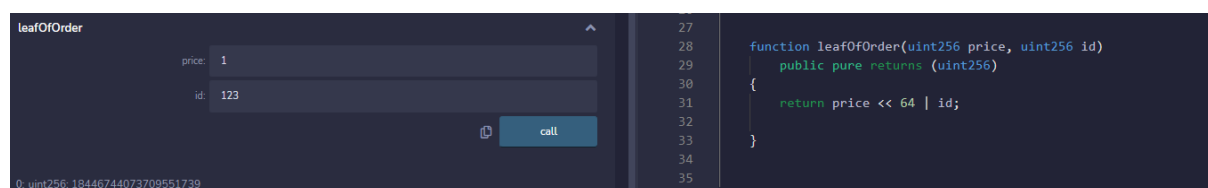
Issue ID	#nft5-001
Category	SWC-101 - Integer Overflow and Underflow
Description	The two functions <i>leafOfOrder</i> and <i>orderIdFromLeaf</i> are designed with bitwise operations, including SHIFT LEFT («), OR () and AND &. However, there is no condition check for the value of <i>order.price</i> . When the value of <i>order.price</i> exceeds 2^{192} , it leads to wrong calculation in <i>leafOfOrder</i> and <i>orderIdFromLeaf</i> .
Severity	MEDIUM
Location(s)	MarketPlace.sol: 333-343
Status	Resolved
Reference	CWE-682 - Incorrect Calculation
Remediation	Have a require statement to control the value of <i>order.price</i> . When it exceeds 2^{192} , return an error message.

Description

When the actual price ***order.price* = $2^{192} + 1 = 6277101735386680763835789423207666416102355444464034512897$** and ***order.id* = 123**, the value returned from *leafOfOrder* is **18446744073709551739**:



However, when both ***order.price* = 1** and ***order.id* = 123**, the value returned from *leafOfOrder* is also **18446744073709551739**:



The returns from **leafOfOrder** are the same for both cases. The price with the value of $2^{192} + 1$ will be sorted before other prices (those with values larger than 1). The contract will not correctly operate under these situations.

The codelines where the issue occurs:

```
...
333     function leafOfOrder(Order memory order)
334         internal pure returns (uint256)
335     {
336         return order.price << 64 | order.id;
337     }
338
339     function orderIdFromLeaf(uint256 leaf)
340         internal pure returns (uint256)
341     {
342         return leaf & ((uint256(1) << 64) - 1);
343     }
...
```

The codebase can be improved as following:

```
...
333     function leafOfOrder(Order memory order)
334         internal pure returns (uint256)
335     {
336         require(order.price < (uint256(1)) << 192, "Value exceeded.");
337         return order.price << 64 | order.id;
338     }
339
340     function orderIdFromLeaf(uint256 leaf)
341         internal pure returns (uint256)
342     {
343         return leaf & ((uint256(1) << 64) - 1);
344     }
...
```

2. Uncontrolled value of *order.id* leads to malfunctioned *orderIdFromLeaf*

Issue ID	#nft5-002
Category	SWC-101 - Integer Overflow and Underflow
Description	The two functions <i>leafOfOrder</i> and <i>orderIdFromLeaf</i> are designed with bitwise operations, including SHIFT LEFT («), OR () and AND &. However, there is no condition check for the value of <i>order.price</i> . When the value of <i>order.id</i> exceeds 2^{64} , it leads to wrong calculation in <i>leafOfOrder</i> and <i>orderIdFromLeaf</i> .
Severity	MINOR
Location(s)	MarketPlace.sol: 333-343
Status	Resolved
Reference	CWE-682 - Incorrect Calculation
Remediation	Due to the nature of <i>order.id</i> , by which means the value of <i>order.id</i> cannot be set manually, we suggest NFT5 design another way to calculate the leaf to avoid this issue.

Description

When the actual id **$order.id = 2^{64} + 1 = 1844674425817699235309551617$** , the value returned from ***leafOfOrder*** is **2268949521066274848769**. However, when we put this value as a parameter for ***orderIdFromLeaf***, the return is **1**:

The screenshot displays a Solidity IDE with two function calls and their corresponding code snippets. On the left, the **leafOfOrder** function is called with `price: 123` and `id: 18446744073709551617`, resulting in `0: uint256: 2268949521066274848769`. Below it, the **orderIdFromLeaf** function is called with `leaf: 2268949521066274848769`, resulting in `0: uint256: 1`. On the right, the code for these functions is shown:

```

26
27
28     function leafOfOrder(uint256 price, uint256 id)
29         public pure returns (uint256)
30     {
31         return price << 64 | id;
32     }
33
34
35
36
37
38     function orderIdFromLeaf(uint256 leaf)
39         public pure returns (uint256)
40     {
41         return leaf & ((uint256(1) << 64) - 1);
42     }
43
44

```

The contract will not correctly operate under any cases, in which the value of ***order.id*** exceeds 2^{64} .

Conclusion

CyStack had conducted a security audit for NFT5's smart contracts. Total 2 issues were found, but none of these issues represented actual bugs or security problems. After a re-test on the new codebase for NFT5's smart contracts, CyStack confirmed that all accepted issues were resolved. No new issues were found for the additional functions in the smart contracts. Overall, the audited smart contracts have 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	<p>Issues, considered as critical, are straightforwardly exploitable bugs and security vulnerabilities.</p> <p>It is advised to immediately resolve these issues in order to prevent major problems or a full failure during contract system operation.</p>
MAJOR	<p>Major issues are bugs and vulnerabilities, which cannot be exploited directly without certain conditions.</p> <p>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.</p>
MEDIUM	<p>In terms of medium issues, bugs and vulnerabilities exist but cannot be exploited without extra steps such as social engineering.</p> <p>It is advised to form a plan of action and patch after high-priority issues have been resolved.</p>
MINOR	<p>Minor issues are generally objective in nature but do not represent actual bugs or security problems.</p> <p>It is advised to address these issues, unless there is a clear reason not to.</p>
INFO	<p>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 makes effective and secure improvements to the smart contract codebase.</p>

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 (<i>external</i> , <i>public</i> , <i>internal</i> or <i>private</i>) is appropriate for a function. By default, functions without concrete specifiers are <i>public</i> .
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 a vulnerable version.
SWC-108	State Variable Default Visibility	Variables can be specified as being <i>public</i> , <i>internal</i> or <i>private</i> . 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 <i>constructor</i>).
	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 <i>delegatecall</i> 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 <i>ecrecover()</i> .

SWC-130	Right-To-Left-Override control character (U+202E)	The character <i>U+202E</i> 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. <i>call()</i>), 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 <i>assert()</i> is actually an invariant. If not, replace the <i>assert()</i> statement with a <i>require()</i> 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 <code>abi.encodePacked()</code> , it's crucial to ensure that a matching signature cannot be achieved using different parameters. Alternatively, you can simply use <code>abi.encode()</code> 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 <i>memory</i> , else <i>storage</i> upon declaration. As of compiler version 0.5.0 and higher this issue has been systematically resolved.
SWC-114	Transaction Dependence Order	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	<code>tx.origin</code> should not be used for authorization. Use <code>msg.sender</code> 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 <i>transfer()</i> and <i>send()</i> and do not otherwise specify a fixed amount of gas when performing calls. Use <i>.call.value(...)(<i>""</i>)</i> 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).