CyStack

CONFIDENTIAL

CYBERSECURITY AUDIT REPORT

Version v1.0

This document details the process and results of the smart contract audit performed by CyStack from 04/05/2022 to 03/06/2022.

Audited for

Coherence Finance

Audited by

Vietnam CyStack Joint Stock Company

Contents

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 Coherence Finance 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	03/06/2022	The first report is sent to the client. All findings are in the open status.

Contact Information

Company	Representative	Position	Email address
Coherence Finance	Andrew Gunderman	Co-founder	andrew@coherence.finance
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

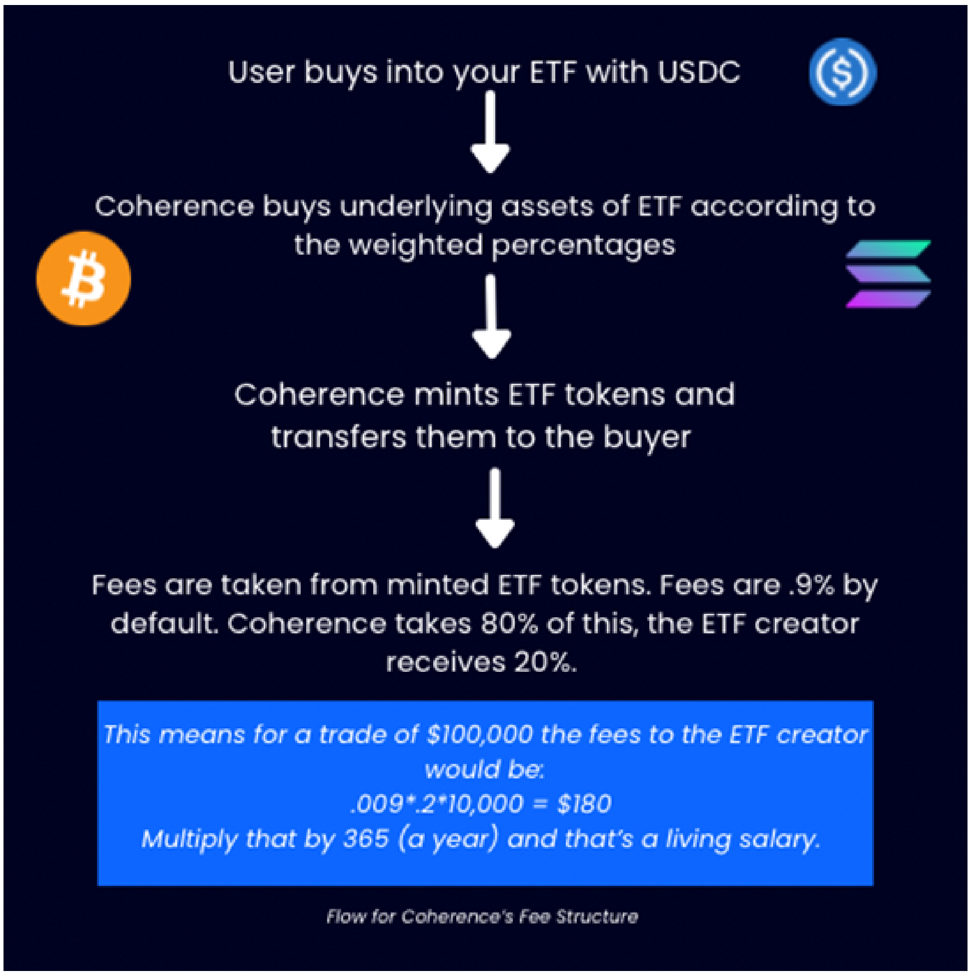
From 04/05/2022 to 03/06/2022, Coherence Finance engaged CyStack to evaluate the security posture of the BeamSplitter of their contract system. Our findings and recommendations are detailed here in this initial report.

1.1 Audit Details

Audit Target

Coherence is a platform on Solana for buying, selling, and building crypto ETFs, which are bundles of many crypto assets. ETFs built on Coherence can contain tens of thousands of assets. These crypto bundles can be either sponsored by Coherence, built by other parties or customized by the platform's users. Especially, the Coherence platform allows ETF creators receive a portion of transaction fees as income. Also, they can pay a listing fee to have their ETF listed higher up on the public ETF list in order to to promote their ETF branding.

The flow for Coherence's fee scheme is illustrated in the following diagram:

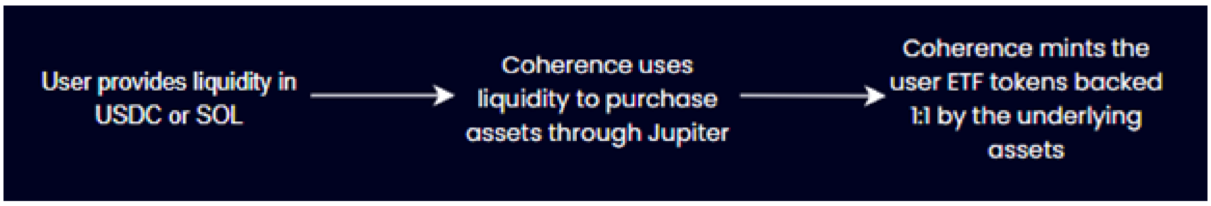

User buys into your ETF with USDC
Coherence buys underlying assets of ETF according to the weighted percentages
Coherence mints ETF tokens and transfers them to the buyer
Fees are taken from minted ETF tokens. Fees are .9% by default. Coherence takes 80% of this, the ETF creator receives 20%.
This means for a trade of $100,000 the fees to the ETF creator would be:
.009*.2*10,000 = $180
Multiply that by 365 (a year) and that's a living salary.
Flow for Coherence's Fee Structure

All of the on-chain functionality for buying, selling, and creating ETFs are controlled by the BeamSplitter program. This program is built with Anchor, which is a framework that simplifies development on Solana.

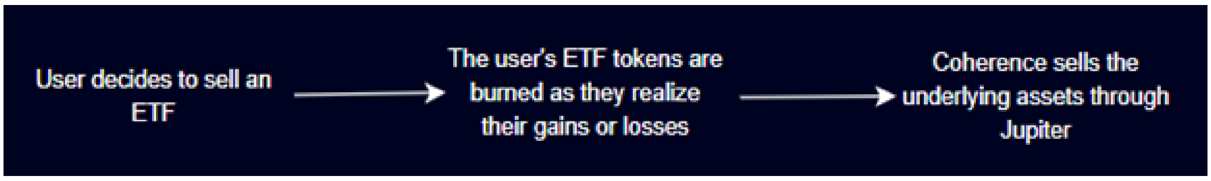
The basic information of BeamSplitter is detailed in the following table:

Item	Description
Project Name	BeamSplitter
Issuer	Coherence Finance
Website	https://coherence.finance/
Platform	Solana Smart Contract
Language	Rust
Codebase	https://github.com/coherence-finance/beamsplitter/tree/master/programs/coherence-beamsplitter/src
Commit	c6e25f18da17c7a15aa84536bffe1cc58d803957
Audit method	Whitebox

From the whitepaper from Coherence, every ETF on this platform is backed 1:1. When a user invest in an ETF, they already have bought each asset in that ETF. The Coherence platform accepts USDC or SOL from users for ETF purchase. Received USDC and SOL are used as liquidity to purchase the underlying assets of the desired ETF through Jupiter Exchange. Hence, only the coins, which are available on Jupiter Exchange, are supported on the Coherence platform. After the demanded assets are successfully bought, an ETF SPL token is minted and transferred to that user. This token represents the exact amount of backing that a user is providing. Because Coherence ETFs are SPL tokens, users can make ETF transactions just like any other SPL tokens.


User provides liquidity in USDC or SOL
Coherence uses liquidity to purchase assets through Jupiter
Coherence mints the user ETF tokens backed 1:1 by the underlying assets

When a user wants to redeem back the original tokens defined in an ETF, or sell an ETF, it is burnt before its underlying assets are transferred to their wallet.


User decides to sell an ETF
The user's ETF tokens are burned as they realize their gains or losses
Coherence sells the underlying assets through Jupiter

In this SCA, as requested by Coherence Finance, CyStack focuses only on finding security issues in the Solana programs written for BeamSplitter, which respectively are:

- **context.rs**: In this file, structs for PrismEtf initialization, finalization, Order start, state, close, Cohere, Decohere, Construction and Deconstruction settings are defined. Also, in context.rs structs for role assignments are defined.
- **enums.rs**: Status codes and types for built PrismEtf, Order and Schedule are defined in this file.
- **errors.rs**: Error codes are defined in this file.
- **lib.rs**: This is the main file, where the logical core of the program is written. This file will make calls to the other files when the program BeamSplitter runs. How PrismEtfs are initialized and finalized file were implemented in lib.rs. Besides, lib.rs decides how to construct and decontruct orders in BeamSplitter. The transaction fees are ruled by lib.rs. By lib.rs, BeamSplitter program is initialized with desired states. Weighted tokens will be set as assets of an ETF, and the ETF then will be minted for cho owners and managers.
- **state.rs**: The definition of basic objects in BeamSplitter are stated in this file.

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. for contracts written in Solidity and libFuzzer, cargo-fuzz, cargo tarpaulin, cargo audit, cargo geiger, Clippy, prusti, etc. for contracts written in Rust. 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 for Solidity smart contracts, 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.

Additionally, SCAs for smart contracts written in Rust are conducted, a following security checklist based on vulnerabilities and advisories gathered from RustSec Advisory Database, the article about Solana common pitfalls from Neodyme Audit Team and the repository Sealevel Attacks.

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 E provides an overview on specific similar software bugs that occur in Smart Contract coding.

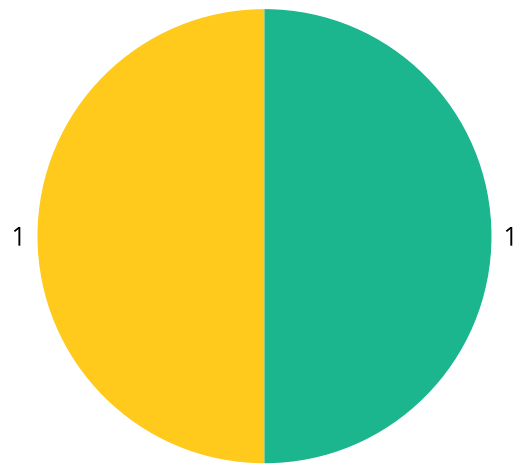
The final report will be sent to the smart contract issuer with an executive summary for overview and detailed results for acts of remediation.

1.4 Audit Scope

Assessment	Target	Type
White-box testing	context.rs	Rust code file
White-box testing	enums.rs	Rust code file
White-box testing	errors.rs	Rust code file
White-box testing	lib.rs	Rust code file
White-box testing	state.rs	Rust code file

Executive Summary

Security issues by severity



Legend

🟥	Critical
🟥	Major
🟧	Medium
🟨	Minor
🟩	Info

Security issues related to Rust smart contracts

Misuse of mathematical expressions	1	🟩
Missing signer check	1	🟨

Security issues by CWE

Protection Mechanism Failure (CWE-693)	1	🟨
Improper Adherence to Coding Standards (CWE-703)	1	🟩

Table of security issues

ID	Status	Vulnerability	Severity
#coherence-001	Open	Missing signer authorization	MINOR
#coherence-002	Open	Absurd comparisons	INFO

Recommendations

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

Key recommendations	
Issues	CyStack conducted security audit for BeamSplitter from Coherence Finance. No issues with severity higher than Medium had been found. Total two issues were found, half of them were related to missing value validation, the other half were coding style defects that increase gas consumption.
Recommendations	Coherence Finance should resolve all the issues in the report unless there is a clear reason not to. CyStack recommends Coherence Finance to evaluate the audit results with several different security audit third-parties for the most accurate conclusion.
References	• https://rustsec.org/ • https://blog.neodyme.io/posts/solana_common_pitfalls • https://github.com/project-serum/sealevel-attacks

Detailed Results

1. Missing signer check

Issue ID	#coherence-001
Category	Missing signer check
Description	A new type **Signer<'info>** is introduced, which validates the account is a signer. For the attributes **manager**, **owner**, **new_owner** and **new_manager** in, respectively, lines 322, 324, 372 and 444 in **context.rs**, they should be defined as **Signer**.
Severity	MINOR
Location(s)	context.rs: 322, 324, 372, 444
Status	Open
Reference	CWE-693 - Protection Mechanism Failure
Remediation	It is recommended to assign the mentioned attributes as **Signer** instead of **AccountInfo**.

Description

The code where the issues occur are lied on the lines 322, 324, 372 and 444 the file context.rs:

```
 …
314  #[derive(Accounts)]
315  pub struct FinalizeOrder<'info> {
316      #[account(mut)]
317      pub prism_etf_mint: Account<'info, Mint>,
318
319      /// The [Signer] of the tx and owner of the [Deposit] [Account]
320      pub orderer: Signer<'info>,
321
322      pub manager: AccountInfo<'info>,
323
324      pub owner: AccountInfo<'info>,
 …
368  }
 …
```

```
…
#[derive(Accounts)]
pub struct SetOwner<'info> {
    pub new_owner: AccountInfo<'info>,

    pub owner: Signer<'info>,
…
}
…
#[derive(Accounts)]
pub struct SetManager<'info> {
    pub prism_etf_mint: Account<'info, Mint>,

    pub new_manager: AccountInfo<'info>,

    pub manager: Signer<'info>,
…
}
…
```

It is recommended to use Signer instead of AccountInfo for these attributes:

```
…
#[derive(Accounts)]
pub struct FinalizeOrder<'info> {
…
    pub manager: Signer<'info>,

    pub owner: Signer<'info>,
…
}

#[derive(Accounts)]
pub struct SetOwner<'info> {
    pub new_owner: Signer<'info>,
…
}
…
#[derive(Accounts)]
pub struct SetManager<'info> {
…
    pub new_manager: Signer<'info>,
…
}
…
```

2. Absurd comparisons

Issue ID	#coherence-002
Category	Misuse of mathematical expressions
Description	In the functions **push_tokens** and **start_order**, comparative operation "less or equal" (<=) to 0 is applied on 64-bit unsigned integer (u64) variables. This comparison involves a case that is always false. The whole expression might misleading imply that it is possible for the unsigned integer variables to be less than 0.
Severity	INFO
Location(s)	lib.rs: 146, 209
Status	Open
Reference	CWE-703 - Improper Adherence to Coding Standards
Remediation	It is recommended to change these expression from **x <= 0** to **x==0**.

Description

The codelines where the issues occur are lied in the file **lib.rs**:

```
…
146             if weighted_token.weight <= 0 {
147                 return Err(BeamsplitterErrors::ZeroWeight.into());
148             }
…
209         if amount <= 0 {
210             return Err(BeamsplitterErrors::ZeroOrder.into());
211         }
…
```

The variable **amount** is defined in the function **start_over** in **lib.rs**:

```
…
188     pub fn start_order(ctx: Context<StartOrder>, order_type: OrderType, amount: u64)
        ↪   -> Result<()> {
…
```

The variable **weighted_token.weight** is the attribute **weight** in an instance of the struct **WeightedToken** defined in **state.rs**:

```
…
#[zero_copy]
#[derive(Debug, Default, AnchorDeserialize, AnchorSerialize)]
pub struct WeightedToken {
    pub mint: Pubkey,
    pub weight: u64,
}

```

It is recommended to change these expression to:

```
…
            if weighted_token.weight == 0 {
                return Err(BeamsplitterErrors::ZeroWeight.into());
            }
…
        if amount == 0 {
            return Err(BeamsplitterErrors::ZeroOrder.into());
        }
…
```

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 makes 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 a vulnerable version.
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).