Blockchain Security & Audits

Seculite

Blockchain Security Company

ETCMC Farming Audit

Security Assessment

21 February 2024

For

ETCMC

Seculite.net

SeculiteAudits

Disclaimer

Seculite Solutions reports are not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. These reports are not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team. Seculite Solutions do not cover testing or auditing the integration with external contracts or services (such as Unicrypt, Uniswap, PancakeSwap etc’...)

Seculite Solutions do not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technology proprietors. Seculite Solutions should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

Seculite Solutions Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Blockchain technology and cryptographic assets present a high level of ongoing risk. Seculite Solutions’ position is that each company and individual are responsible for their own due diligence and continuous security. Seculite Solutions in no way claims any guarantee of the security or functionality of the technology we agree to analyze. Thus, the audit can’t guarantee explicit security of the audited smart contracts.

Version	Date	Description
1.0	21 February 2024	• Layout project • Automated- /Manual-Security Testing • Summary

Network

Ethereum Classic (ETC)

Website

https://etc-mc.com

Description

ETCMC is aiming to be one of the decentralised exchanges (DEX) with an automated market-maker (AMM) on the Ethereum Classic Blockchain.

Project Engagement

During the Date of 14 February 2024, **ETCMC Team** engaged Seculite to audit smart contracts that they created. The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts. They provided Seculite with access to their code repository.

Logo

Contract Link

v1.0
• Provided as files

https://github.com/etcmc-org/etcmc_farming_smartcontract

Disclaimer 2

Description 4
Project Engagement 4
Logo 4
Vulnerablity & Risk Level 6
Audit Strategy and Techniques Applied 7
Methodology 7
Used code from other Frameworks/Smart Contracts 8
Tested Contract Files 9
Source Lines 10
Components/Exposed Functions 11
Inheritance Graph 12
Call Graph 13
Scope of Work/Verify Claims 14
Overall checkup(Smart contract Security) 15
Modifiers and public functions 16
Ownership Privileges 17
Source Units in Scope 18

Audit Results 19
Critical Issues 19
High Issues 19
Medium Issues 19
Low Issues 19
Audit Comments 20
SWC Attacks 21

Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.0.

Level	Value	Vulnerability	Risk (Required Action)
Critical	9 - 10	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
High	7-8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
Medium	4-6.9	vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	implementation of corrective actions in a certain period.
Low	2-3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	implementation of certain corrective actions or accepting the risk.
Informational	0-1.9	A vulnerability that have informational character but is not effecting any of the code.	An observation that does not determine a level of risk.

Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. To do so, reviewed line-by-line by our team of expert pen testers and smart contract developers, documenting any issues as there were discovered.

Methodology

The auditing process follows a routine series of steps:

1. Code review that includes the following:
    a. Review of the specifications, sources, and instructions provided to Seculite Solution to make sure we understand the size, scope, and functionality of the smart Contract.
    b. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
    c. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Seculite Solution describe.
2. Testing and automated analysis that includes the following:
    a. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
    b. Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemised, actionable recommendations to help you take steps to secureyour smart contracts.

Used Code from other Frameworks/Smart Contracts (direct imports)

Imported Packages:

Dependency / Import Path	Count
@openzeppelin/contracts/math/Math.sol	1
@openzeppelin/contracts/math/SafeMath.sol	1
@openzeppelin/contracts/ownership/Ownable.sol	1
@openzeppelin/contracts/token/ERC20/ERC20Detailed.sol	1
@openzeppelin/contracts/token/ERC20/IERC20.sol	1
@openzeppelin/contracts/token/ERC20/SafeERC20.sol	1
@openzeppelin/contracts/utils/ReentrancyGuard.sol	1

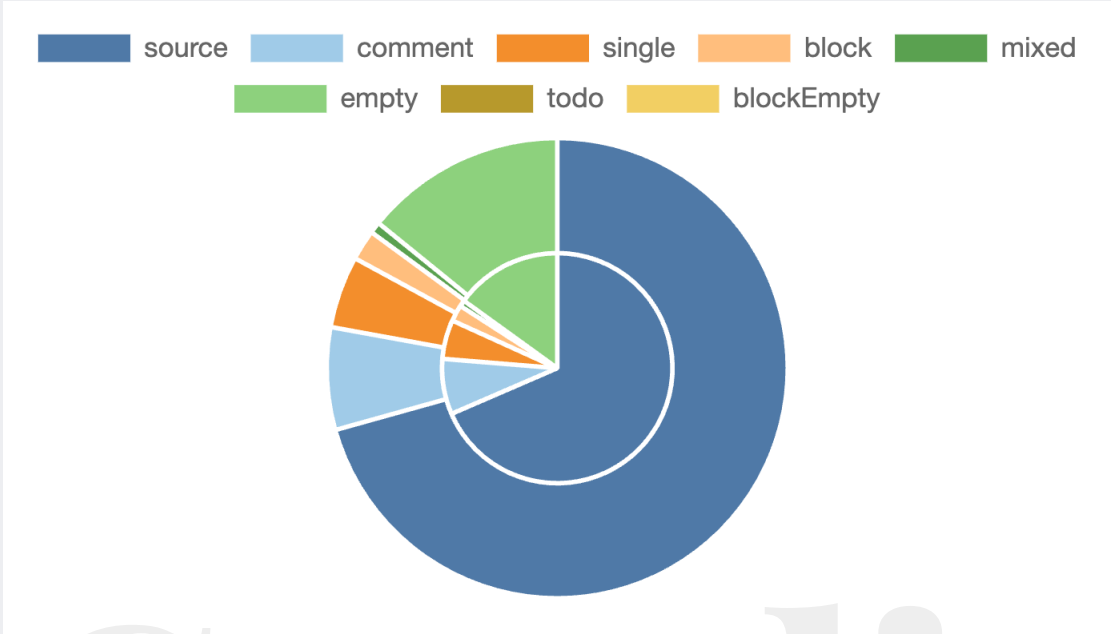
Tested Contract Files

This audit covered the following files listed below with a SHA-1 Hash.
A file with a different Hash has been modified, intentionally or otherwise, after the security review. A different Hash could be (but not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of this review.

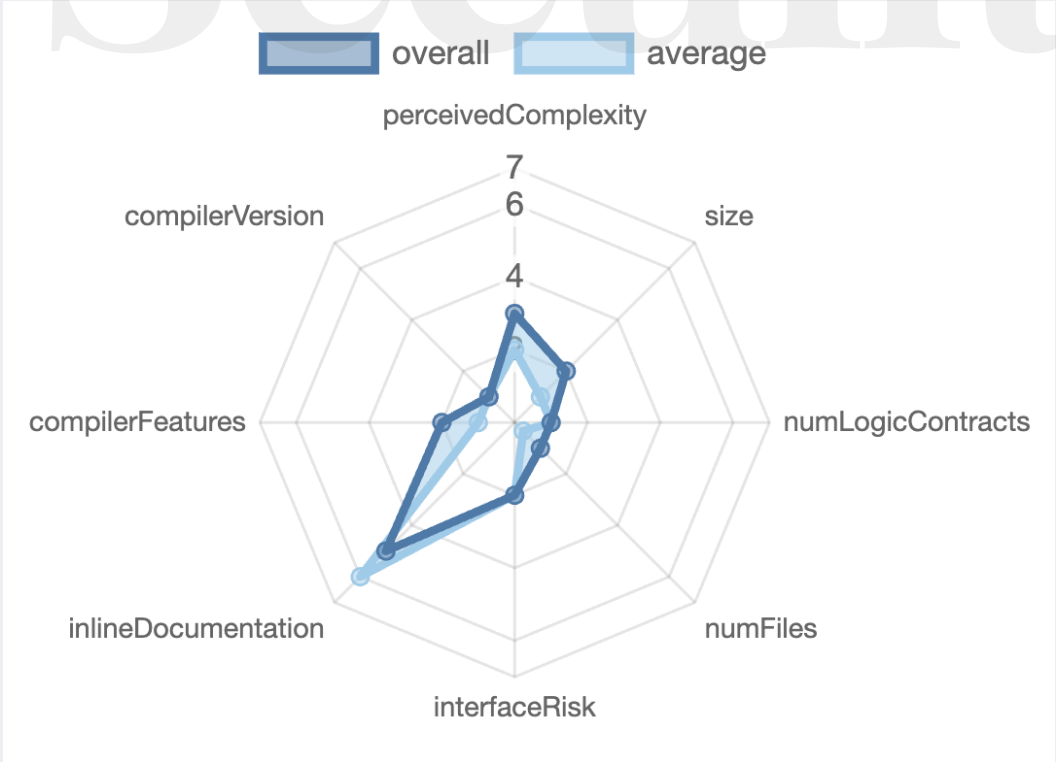
File Name	SHA-1 Hash
contracts/RewardsDistributionRecipient.sol	66ed36006b452f624a495d7487d34382669c77f4
contracts/StakingRewards.sol	9d3733aacfcb027040d41066748298cce0bda0b1
contracts/StakingRewardsFactory.sol	a665672ea595427a4d5a4fa621ab767584729717

Metrics

Source Lines: v1.0



Risk Level: v1.0



Capabilities

Components

📝Contracts	📚Libraries	🔍Interfaces	🎨Abstract
3	0	1	0

Exposed Functions

This section lists functions that are explicitly declared public or payable. Please note that getter methods for public stateVars are not included.

🌐Public	💰Payable
19	0

External	Internal	Private	Pure	View
9	13	0	0	6

State Variables

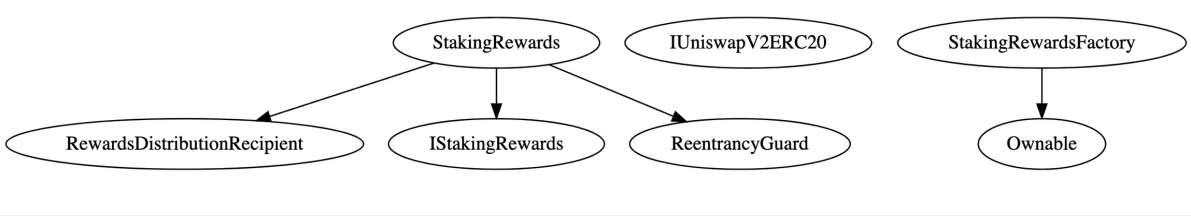
Total	🌐Public
16	14

Capabilities

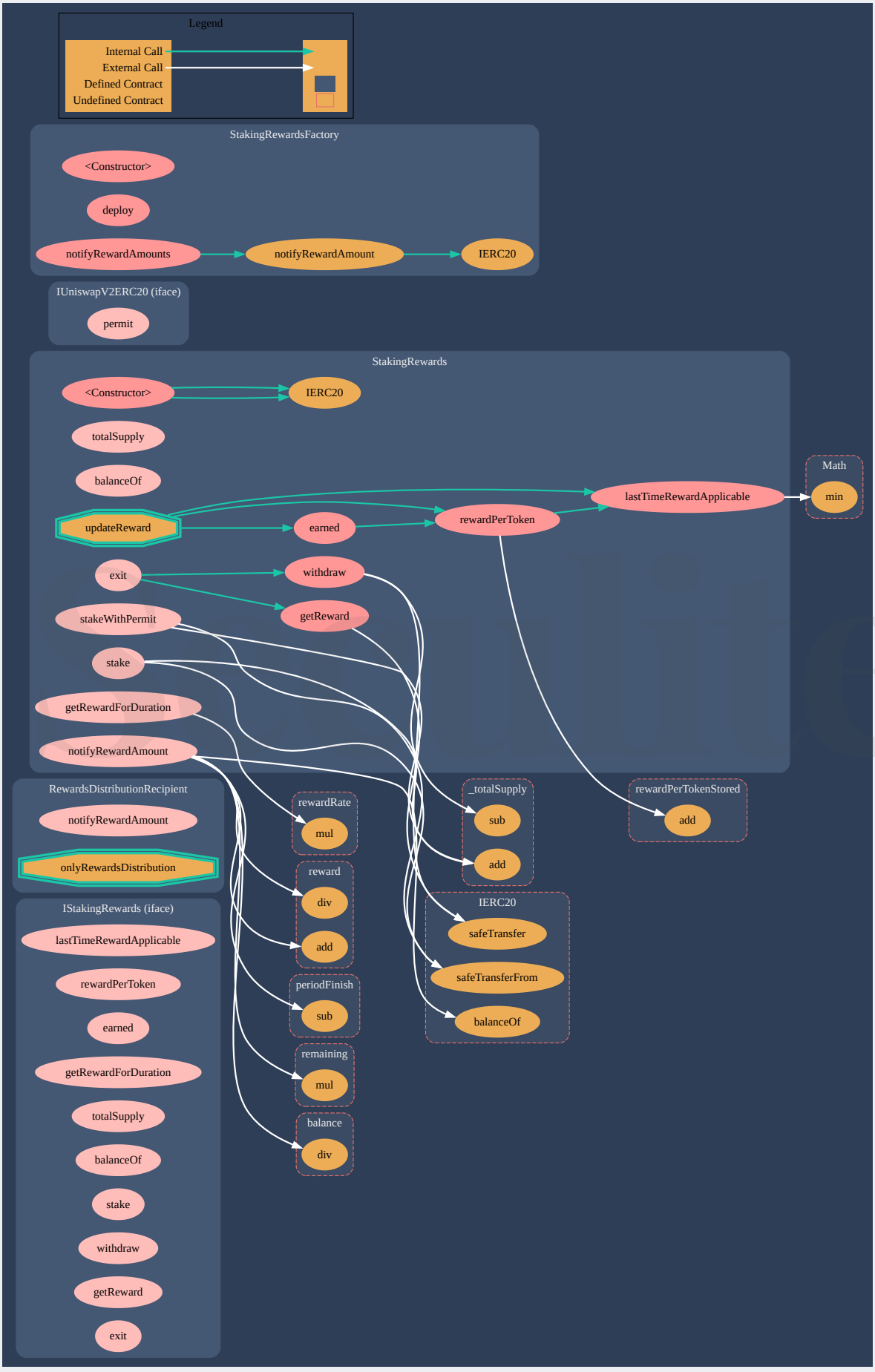
Solidity Versions observed	🧪 Experimental Features	💰 Can Receive Funds	🖥️ Uses Assembly	💣 Has Destroyable Contracts
0.5.16				

📩 Transfers ETH	⚡ Low-Level Calls	👥 DelegateCall	🧱 Uses Hash Functions	🪚 ECRecover	🌀 New/Create/Create2
yes					yes → NewContract:StakingRewards

Inheritance Graph



CallGraph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
StakingRewardsFactory
<Constructor>
deploy
notifyRewardAmounts
notifyRewardAmount
IERC20
IUniswapV2ERC20 (iface)
permit
StakingRewards
<Constructor>
IERC20
totalSupply
balanceOf
updateReward
earned
rewardPerToken
lastTimeRewardApplicable
Math
min
exit
withdraw
getReward
stakeWithPermit
stake
getRewardForDuration
notifyRewardAmount
RewardsDistributionRecipient
notifyRewardAmount
onlyRewardsDistribution
IStakingRewards (iface)
lastTimeRewardApplicable
rewardPerToken
earned
getRewardForDuration
totalSupply
balanceOf
stake
withdraw
getReward
exit
rewardRate
mul
reward
div
add
periodFinish
sub
remaining
mul
balance
div
_totalSupply
sub
add
IERC20
safeTransfer
safeTransferFrom
balanceOf
rewardPerTokenStored
add

Scope of Work/Verify Claims

The above token Team provided us with the files that needs to be tested(Github, Bscscan, Etherscan, les, etc.). The scope of the audit is the main contract (usual the same name as team appended with .sol).

We will verify the following claims:

1. Overall checkup (Smart Contract Security)

Overall checkup (Smart Contract Security)

Tested	Verified
✓	✓

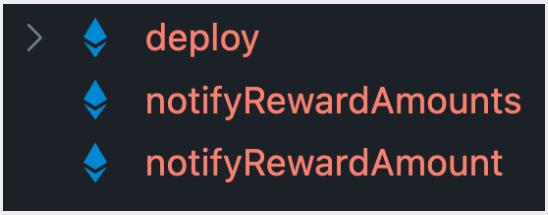
Legend

Attribute	Symbol
Verified / Checked	✓
Parity Verified	⚑
Unverified / Not Verified	✗
Not available	-

Modifiers and public functions

StakingRewards.sol

StakingRewardsFactory.sol



Note:

- General fork from Uniswap/liquidity-staker

Ownership Privileges:

- The owner is also able to deploy a new staking pool
- The owner can’t mint, burn or withdraw tokens

Source Units in Scope v1.0

Type	File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines	Complex. Score	Capabilities
	contracts/StakingRewardsFactory.sol	1		110	110	82	12	51	
	contracts/StakingRewards.sol	1	1	217	190	144	15	117	
	contracts/RewardsDistributionRecipient.sol	1		15	13	11		5	
	Totals	**3**	**1**	**342**	**313**	**237**	**27**	**173**	

Attribute	Description
Lines	total lines of the source unit
nLines	normalised lines of the source unit (e.g. normalises functions spanning multiple lines)
nSLOC	normalised source lines of code (only source-code lines; nocomments, no blank lines)
Comment Lines	lines containing single or block comments
Complexity Score	a custom complexity score derived from code statements that are known to introduce code complexity(branches,loops,,calls,external interfaces, ...)

Audit Results

Critical issues

No critical issues

High issues

No high issues

Medium issues

No medium issues

Low issues

No low issues

Informational

In the upcoming scenario, it is indicated that the loop could encounter failure. It is essential to conduct tests using the highest possible levels, following the established plan.

```
// call notifyRewardAmount for all staking tokens.
function notifyRewardAmounts() public {   infinite gas
    require(
        stakingTokens.length > 0,
        "StakingRewardsFactory::notifyRewardAmounts: called before any deploys"
    );
    for (uint i = 0; i < stakingTokens.length; i++) {
        notifyRewardAmount(stakingTokens[i]);
    }
}
```

Audit Comments

We recommend you to use the special form of comments (NatSpec Format, Follow link for more information https://docs.soliditylang.org/en/ latest/natspec-format.html) for your contracts to provide rich documentation for functions, return variables and more.

This helps investors to make clear what that variables, functions etc. do.

SWC Attacks

ID	TITLE	RELATIONSHIPS	STATUS
SWC-136	Unencrypted Private Data On-Chain	CWE-767: Access to Critical Private Variable via Public Method	PASSED
SWC-135	Code With No Effects	CWE-1164: Irrelevant Code	PASSED
SWC-134	Message call with hardcoded gas amount	CWE-655: Improper Initialization	PASSED
SWC-133	Hash Collisions With Multiple Variable Length Arguments	CWE-294: Authentication Bypass by Capture-replay	PASSED
SWC-132	Unexpected Ether balance	CWE-667: Improper Locking	PASSED
SWC-131	Presence of unused variables	CWE-1164: Irrelevant Code	PASSED
SWC-130	Right-To-Left-Override control character (U+202E)	CWE-451: User Interface (UI) Misrepresentation of Critical Information	PASSED
SWC-129	Typographical Error	CWE-480: Use of Incorrect Operator	PASSED
SWC-128	DoS With Block Gas Limit	CWE-400: Uncontrolled Resource Consumption	PASSED
SWC-127	Arbitrary Jump with Function Type Variable	CWE-695: Use of Low-Level Functionality	PASSED
SWC-126	Insufficient Gas Griefing	CWE-691: Insufficient Control Flow Management	PASSED
SWC-125	Incorrect Inheritance Order	CWE-696: Incorrect Behavior Order	PASSED
SWC-124	Write to Arbitrary Storage Location	CWE-123: Write-what-where Condition	PASSED
SWC-123	Requirement Violation	CWE-573: Improper Following of Specification by Caller	PASSED
SWC-122	Lack of Proper Signature Verification	CWE-345: Insufficient Verification of Data Authenticity	PASSED
SWC-121	Missing Protection against Signature Replay Attacks	CWE-347: Improper Verification of Cryptographic Signature	PASSED
SWC-120	Weak Sources of Randomness from Chain Attributes	CWE-330: Use of Insufficiently Random Values	PASSED
SWC-119	Shadowing State Variables	CWE-710: Improper Adherence to Coding Standards	PASSED
SWC-118	Incorrect Constructor Name	CWE-665: Improper Initialization	PASSED
SWC-117	Signature Malleability	CWE-347: Improper Verification of Cryptographic Signature	PASSED
SWC-116	Block values as a proxy for time	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	PASSED
SWC-115	Authorization through tx.origin	CWE-477: Use of Obsolete Function	PASSED
SWC-114	Transaction Order Dependence	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	PASSED
SWC-113	DoS with Failed Call	CWE-703: Improper Check or Handling of Exceptional Conditions	PASSED
SWC-112	Delegatecall to Untrusted Callee	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	PASSED
SWC-111	Use of Deprecated Solidity Functions	CWE-477: Use of Obsolete Function	PASSED
SWC-110	Assert Violation	CWE-670: Always-Incorrect Control Flow Implementation	PASSED
SWC-109	Uninitialized Storage Pointer	CWE-824: Access of Uninitialized Pointer	PASSED
SWC-108	State Variable Default Visibility	CWE-710: Improper Adherence to Coding Standards	PASSED
SWC-107	Reentrancy	CWE-841: Improper Enforcement of Behavioral Workflow	PASSED
SWC-106	Unprotected SELFDESTRUCT Instruction	CWE-284: Improper Access Control	PASSED
SWC-105	Unprotected Ether Withdrawal	CWE-284: Improper Access Control	PASSED
SWC-104	Unchecked Call Return Value	CWE-252: Unchecked Return Value	PASSED
SWC-103	Floating Pragma	CWE-664: Improper Control of a Resource Through its Lifetime	PASSED
SWC-102	Outdated Compiler Version	CWE-937: Using Components with Known Vulnerabilities	PASSED
SWC-101	Integer Overflow and Underflow	CWE-682: Incorrect Calculation	PASSED
SWC-100	Function Default Visibility	CWE-710: Improper Adherence to Coding Standards	PASSED

A Blockchain Security Company

Seculite Secured

Seculite.net

SeculiteAudits