Cyberscope

Audit Report

Circle Launchpad Locker

March 2022

Github	https://github.com/monkey-shanti/Circle-Launchpad
Commit	7f6f46693c2710c2dfc986618b6531561f0ddabb
Audited by	© cyberscope

Table of Contents

Contract Review

Repository	https://github.com/monkey-shanti/Circle-Launchpad
Commit	7f6f46693c2710c2dfc986618b6531561f0ddabb

Contract Name	CircleLocker
Testing Deploy	https://testnet.bscscan.com/address/0x6D557a6e3A872dB14D981BA1E7268757536EB3Cf

Audit Updates

Initial Audit	20 Dec 2022 https://github.com/cyberscope-io/audits/blob/main/circleLaunchpad/v1/locker.pdf
Corrected Phase 2	02 Jan 2023 https://github.com/cyberscope-io/audits/blob/main/circleLaunchpad/v2/locker.pdf
Corrected Phase 3	16 Mar 2023

Source Files

Filename	SHA256
launchpad/interfaces/IUniswapV2Pair.sol	5631411f67c8741031e9bfdd27fad3c81554c0c92a37dce6990b618ef634cd0a
launchpad/interfaces/PoolLibrary.sol	f209394b1e0c66187d6e6f86f5de7708930fe8f282c3fbfbe69e507dc5133939
launchpad/libraries/LibEnsureSafeTransfer.sol	a4c2990e467b6b694059f106497a2c31f4489c6723fa6470c076a919548cc7ca
Locker.sol	1fc523d7494f0dbaa480c2b2899a893cdab6d95c83cbe4ce14ec556318cbceb3
utils/Utility.sol	bb982ca156ddbd0ea26ba803843a755ff4ad6440addadc577d3ba8335f8e0705

Introduction

The Locker contract implements a locker mechanism.

Roles

The contract has three roles.

Owner Role

The owner role has the authority to WithdrawBNB.

Locker Owner Role

- editLockDescription
- editLock
- withdrawableTokens
- unlock
- transferLockOwnership

User Role

The users have the authority to

- multipleVestingLock
- vestingLock
- lock

Contract Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	LTM	LP Token Mocking	Unresolved
●	LLS	Locker Logic Simplification	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L05	Unused State Variable	Unresolved
●	L08	Tautology or Contradiction	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L12	Using Variables before Declaration	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L19	Stable Compiler Version	Unresolved

LTM - LP Token Mocking

Criticality	Minor / Informative
Location	Locker.sol#L908
Status	Unresolved

Description

The user can create a contract that implements the `factory()`, `getPair()` and `token0()`, `token1()` method in order to mock the LP token validator. As a result the user will be able to lock an LP Token that essentially is not an LP token.

```
function _isValidLpToken(address token, address factory)
private
view
returns (bool)
{
    IUniswapV2Pair pair = IUniswapV2Pair(token);
    address factoryPair = IUniswapV2Factory(factory).getPair(
        pair.token0(),
        pair.token1()
    );
```

LLS - Locker Logic Simplification

Criticality	Minor / Informative
Location	Locker.sol#L24
Status	Unresolved

Description

The normal and vesting lock could be the same since normal lock is equal to the vesting lock with 100% return in the first circle.

```
Normal lock == Vesting with:
tgeBps = 10_000
tgeDate = lock date
cycleBps = lock start date
cycle = 1
```

Recommendation

The contract could merge and simplify the logic of the vesting and normal token lock.

```
struct Lock {
    uint256 id;
    address token;
    address owner;
    uint256 amount;
    uint256 lockDate;
    uint256 tgeDate; // TGE date for vesting locks, unlock date for
normal locks
    uint256 tgeBps; // In bips. Is 0 for normal locks
    uint256 cycle; // Is 0 for normal locks
    uint256 cycleBps; // In bips. Is 0 for normal locks
    uint256 unlockedAmount;
    string description;
    bool isVesting;
}
```

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Locker.sol#L858
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
uint256 _amount
address payable _receiver
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	Locker.sol#L85
Status	Unresolved

Description

An unused state variable is a state variable that is declared in the contract, but is never used in any of the contract's functions. This can happen if the state variable was originally intended to be used, but was later removed or never used.

Unused state variables can create clutter in the contract and make it more difficult to understand and maintain. They can also increase the size of the contract and the cost of deploying and interacting with it.

```
uint256 private constant ID_PADDING = 1_000_000
```

Recommendation

To avoid creating unused state variables, it's important to carefully consider the state variables that are needed for the contract's functionality, and to remove any that are no longer needed. This can help improve the clarity and efficiency of the contract.

L08 - Tautology or Contradiction

Criticality	Minor / Informative
Location	Locker.sol#L183
Status	Unresolved

Description

A tautology is a logical statement that is always true, regardless of the values of its variables. A contradiction is a logical statement that is always false, regardless of the values of its variables.

Using tautologies or contradictions can lead to unintended behavior and can make the code harder to understand and maintain. It is generally considered good practice to avoid tautologies and contradictions in the code.

```
require(tgeBps >= 0, "Invalid bips for TGE")
```

Recommendation

The team is advised to carefully consider the logical conditions is using in the code and ensure that it is well-defined and make sense in the context of the smart contract.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	launchpad/libraries/LibEnsureSafeTransfer.sol#L85,94,102,114
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function safeTransferFrom(
        address token,
        address sender,
        address recipient,
        uint256 amount
    ) internal validAddress(token) validAddress(sender)
validAddress(recipient) validAmount(amount) {
        IERC20Upgradeable(token).safeTransferFrom(sender, recipient,
amount);
    }

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L12 - Using Variables before Declaration

Criticality	Minor / Informative
Location	Locker.sol#L832
Status	Unresolved

Description

The contract is using a variable before the declaration. This is usually happening either if it has not been declared yet or if the variable has been declared in a different scope. It is not a good practice to use a local variable before it has been declared.

```
address factory
```

Recommendation

By declaring local variables before using them, contract ensures that it operates correctly. It's important to be aware of this rule when working with local variables, as using a variable before it has been declared can lead to unexpected behavior and can be difficult to debug.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	Locker.sol#L541
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 currentTotal = (((block.timestamp - userLock.tgeDate) / userLock.cycle) * cycleReleaseAmount) + tgeReleaseAmount
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	Locker.sol#L832
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
address factory
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	launchpad/libraries/LibEnsureSafeTransfer.sol#L1
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.0;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Contract Functions

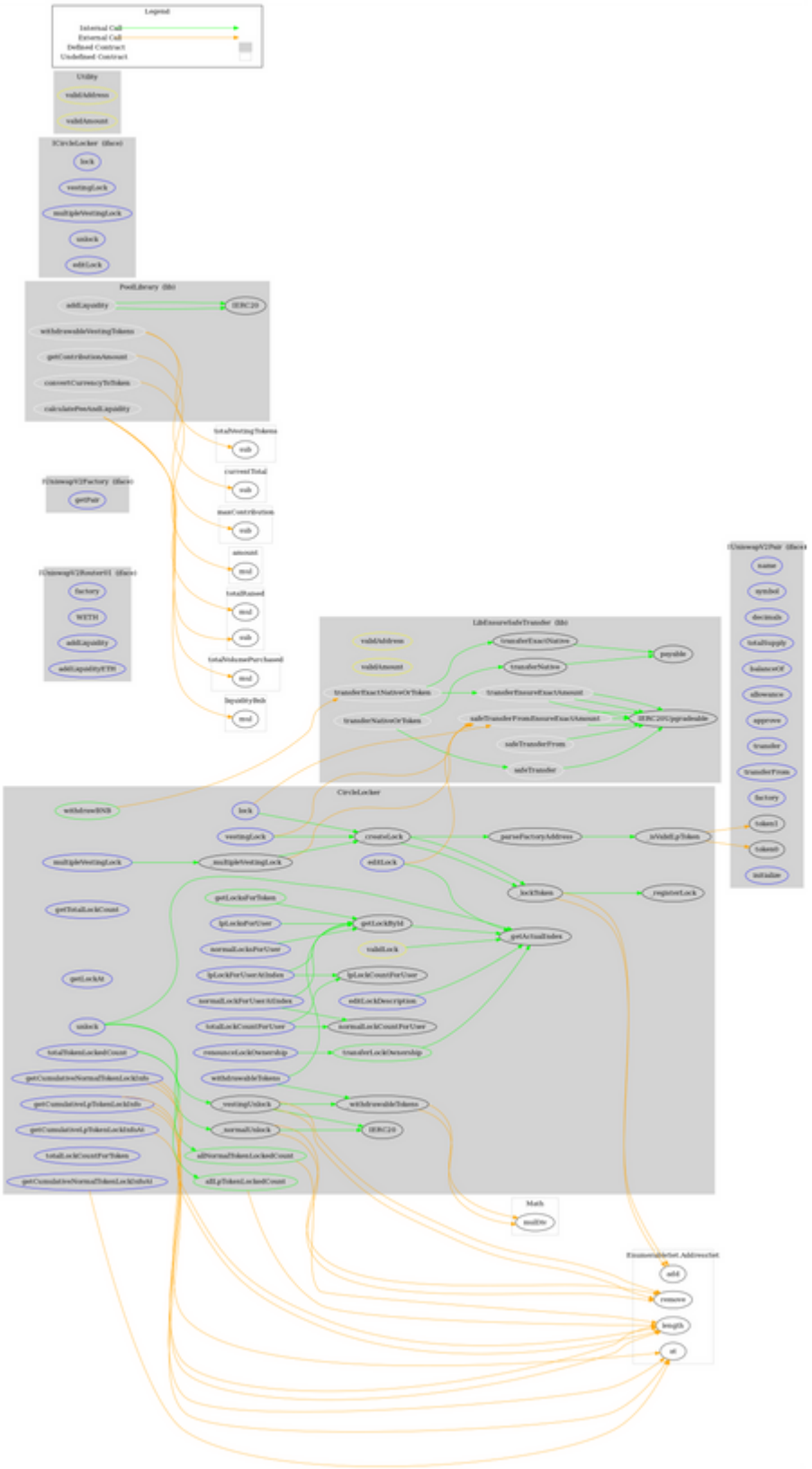
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IUniswapV2Pair	Interface			
	name	External		-
	symbol	External		-
	decimals	External		-
	totalSupply	External		-
	balanceOf	External		-
	allowance	External		-
	approve	External	✓	-
	transfer	External	✓	-
	transferFrom	External	✓	-
	factory	External		-
	token0	External		-
	token1	External		-
	initialize	External	✓	-
IUniswapV2Router01	Interface			
	factory	External		-
	WETH	External		-
	addLiquidity	External	✓	-
	addLiquidityETH	External	Payable	-
IUniswapV2Router02	Interface	IUniswapV2Router01		

IUniswapV2Factory	Interface			
	getPair	External		-
PoolLibrary	Library			
	withdrawableVestingTokens	Internal		
	getContributionAmount	Internal		
	convertCurrencyToToken	Internal		
	addLiquidity	Internal	✓	
	calculateFeeAndLiquidity	Internal		
LibEnsureSafeTransfer	Library			
	safeTransferFromEnsureExactAmount	Internal	✓	validAddress validAddress validAddress validAmount
	transferEnsureExactAmount	Internal	✓	validAddress validAddress validAmount
	transferExactNativeOrToken	Internal	✓	
	transferExactNative	Internal	✓	validAddress validAmount
	safeTransferFrom	Internal	✓	validAddress validAddress validAddress validAmount
	safeTransfer	Internal	✓	validAddress validAddress validAmount
	transferNativeOrToken	Internal	✓	
	transferNative	Internal	✓	validAddress validAmount
ICircleLocker	Interface			
	lock	External	✓	-
	vestingLock	External	✓	-

	multipleVestingLock	External	✓	-
	unlock	External	✓	-
	editLock	External	✓	-
CircleLocker	Implementation	ICircleLocker, Ownable, Utility		
	lock	External	✓	-
	vestingLock	External	✓	-
	multipleVestingLock	External	✓	-
	_multipleVestingLock	Internal	✓	
	_createLock	Internal	✓	
	_lockToken	Private	✓	
	_registerLock	Private	✓	
	unlock	External	✓	validLock
	_normalUnlock	Internal	✓	
	_vestingUnlock	Internal	✓	
	withdrawableTokens	External		-
	_withdrawableTokens	Internal		
	editLock	External	✓	validLock
	editLockDescription	External	✓	validLock
	transferLockOwnership	Public	✓	validLock
	renounceLockOwnership	External	✓	-
	getTotalLockCount	External		-
	getLockAt	External		-
	getLockById	Public		-
	allLpTokenLockedCount	Public		-
	allNormalTokenLockedCount	Public		-
	getCumulativeLpTokenLockInfoAt	External		-
	getCumulativeNormalTokenLockInfoAt	External		-

	getCumulativeLpTokenLockInfo	External		-
	getCumulativeNormalTokenLockInfo	External		-
	totalTokenLockedCount	External		-
	lpLockCountForUser	Public		-
	lpLocksForUser	External		-
	lpLockForUserAtIndex	External		-
	normalLockCountForUser	Public		-
	normalLocksForUser	External		-
	normalLockForUserAtIndex	External		-
	totalLockCountForUser	External		-
	totalLockCountForToken	External		-
	getLocksForToken	Public		-
	_getActualIndex	Internal		
	_parseFactoryAddress	Internal		
	_isValidLpToken	Private		
	withdrawBNB	Public	✓	onlyOwner validAddress validAmount
Utility	Implementation			

Contract Flow



Inheritance Graph

Summary

The Locker contract implements a locker mechanism. This audit investigates security issues, business logic concerns, and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment 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 Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io