Cyberscope

Audit Report

RandFi

Aug 2023

Network	BSC
Address	0x14DFe32aDA1225D1384F8bC59E7527F1C2C907d5
Audited by	© cyberscope

Table of Contents

Review

Explorer	https://bscscan.com/address/0x14dfe32ada1225d1384f8bc59e7527f1c2c907d5

Audit Updates

Initial Audit	09 Aug 2023

Source Files

Filename	SHA256
contracts/Vault.sol	2966fd30ef79e1c326d60d6c1394524c8bf51f0cbbadb9563b7fade7905e3e91
@openzeppelin/contracts/utils/Address.sol	8160a4242e8a7d487d940814e5279d934e81f0436689132a4e73394bab084a6d
@openzeppelin/contracts/token/ERC20/IERC20.sol	94f23e4af51a18c2269b355b8c7cf4db8003d075c9c541019eb8dcf4122864d5
@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol	0c8a43f12ac2081c6194d54da96f02ebc457760d6514f6b940689719fcef8c0a
@openzeppelin/contracts/token/ERC20/extensions/draft-IERC20Permit.sol	3e7aa0e0f69eec8f097ad664d525e7b3f0a3fda8dcdd97de5433ddb131db86ef

Overview

The Vault contract is a secure storage mechanism designed to handle deposits and withdrawals of different tokens. The contract operates with predefined roles, namely the `keeper` and `mkeeper`, which are defined upon deployment. These roles are critical for the operation of the contract, and their addresses can be modified or toggled by a designated `Guard` address.

Roles and Permissions

- keeper: A set of addresses with the privilege to deposit and withdraw tokens.
- mkeeper: keeping track of all addresses that have ever been assigned the `keeper` role. Any address that's added or removed from the keeper role is reflected in the `mkeeper` mapping. However, the `mkeeper` does not grant any specific capabilities or permissions.
- Guard: A special role responsible for administrative functions like updating keepers, toggling the Update state, adjusting the withdrawal limit (`Limit`), and more.

Deposit Mechanism

Users with the `keeper` role can deposit a certain amount of tokens into the contract. If the token has not been added to the vaults mapping yet, its stablecoin counterpart is registered. The `vault` mapping then keeps track of each token's stablecoin address and the total amount deposited.

Withdrawal Mechanism

Users with the `keeper` role can also withdraw a specific amount from the contract. There's a daily withdrawal limit defined by the `Limit` variable, ensuring that the withdrawal amount does not exceed a certain percentage of the total tokens held by the vault contract. The contract uses the `meter` mapping to keep track of the daily withdrawal amounts based on the block's timestamp.

Guard Functions

- The `safe` function can be used to toggle the state of a `keeper` address.

- The `setLimit` function allows the `Guard` to adjust the maximum daily withdrawal `limit`.
- The `updateKeeper` function allows the `Guard` to add or remove addresses from both the `keeper` and `mkeeper` roles.
- The `off` function allows the `Guard` to toggle off the Update state, potentially locking in the list of keepers.

Roles

Guard

The `Guard` is a privileged role that has authority over critical functions in the Vault contract. The Guard is set during the deployment process via the constructor. The responsibilities of the Guard include

- Setting and modifying the daily withdrawal limit using `setLimit`
- Toggling the state of `keeper` addresses with safe.
- Updating `keeper` addresses via `updateKeeper`
- Disabling the updating mechanism with the `off` function.

Keeper

The `keeper` is a role that can perform core operations within the contract. Addresses with the `keeper` role are defined at the time of contract deployment, and they can interact with the following functions:

- Deposit Tokens: Using the `deposit` function, a keeper can deposit a certain amount of tokens to the contract, and if the token hasn't been registered yet, its associated stablecoin address is stored.
- Withdraw Tokens: A keeper can withdraw tokens through the `withdraw` function. This function ensures that the withdrawal respects the daily limit and deducts the tokens from the vault's balance.

Functions

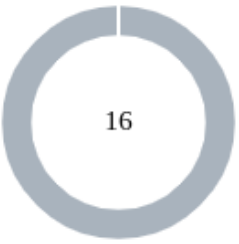
The `keeper` can interact with the following functions:

- `deposit(address token, address stable, uint256 amount)`
- `withdraw(address token, address to, uint256 amount)`
- `getVault(address token)`

The `Guard` can interact with the following functions:

- `safe(address _keeper)`
- `setLimit(uint256 limit)`
- `updateKeeper(address _keeper, bool state)`
- `off()`

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	16

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	16	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	MTV	Missing Transfer Verification	Unresolved
	CO	Code Optimization	Unresolved
	DSO	Data Structure Optimization	Unresolved
	CR	Centralization Risk	Unresolved
	MC	Missing Check	Unresolved
	MU	Modifiers Usage	Unresolved
	RSW	Redundant Storage Writes	Unresolved
	MEE	Missing Events Emission	Unresolved
	IDI	Immutable Declaration Improvement	Unresolved
	L04	Conformance to Solidity Naming Conventions	Unresolved
	L07	Missing Events Arithmetic	Unresolved
	L09	Dead Code Elimination	Unresolved
	L16	Validate Variable Setters	Unresolved
	L17	Usage of Solidity Assembly	Unresolved

●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved

MTV - Missing Transfer Verification

Criticality	Minor / Informative
Location	contracts/Vault.sol#L28
Status	Unresolved

Description

The contract allows deposits of stable tokens by updating the amount in the `vaults variable` for a specific token. The `deposit` function accepts a `token` address, a `stable` token address, and an `amount`. It first checks if the caller is a valid keeper. If the `vaults[token].stable address` is not set, it assigns the provided stable address to it. Subsequently, the function simply increases the vault's amount by the specified deposit `amount`. However, the function assumes the transfer of stable tokens without actually verifying if the tokens from the `stable` address have been transferred to the vault address.

```
    function deposit(address token, address stable, uint256
amount) external {
        require(keeper[msg.sender], "Vault: not keeper");
        if (vaults[token].stable == address(0)) {
            vaults[token].stable = stable;
        }
        vaults[token].amount += amount;
    }
```

Recommendation

It is recommended to incorporate a mechanism that verifies the transfer of `stable` tokens to the `vaults` address. One approach would be to check the balance of the `stable` token for the vault address before and after the deposit action. The difference in balances should match the deposit amount. This ensures that the deposit action genuinely reflects the actual token transfer, providing a more secure and accurate representation of the vault's holdings.

CO - Code Optimization

Criticality	Minor / Informative
Location	contracts/Vault.sol#L36
Status	Unresolved

Description

The contract is structured to allow withdrawals from a vault, with a daily withdrawal limit defined for each token. To track and enforce this daily limit, the contract uses the `meter` mapping which maps days, calculated as `block.timestamp / 1 days`, to the amount withdrawn on that day. However, this approach might be gas inefficient. It is more efficient and clearer to maintain two distinct variables: `meterDay` to store the current day and `meterAmount` to store the amount withdrawn on that day. This structure would allow the contract to easily check if the day has changed (by comparing the current day with `meterDay`), and if so, reset `meterAmount` to zero.

```
    function withdraw(address token, address to, uint256 amount)
external {
        ...
        uint256 ds = block.timestamp / 1 days;
        require(
            meter[ds] + amount <=
                ((meter[ds] + vaults[token].amount) * Limit) /
10000,
            "Vault: limit"
        );
        meter[ds] += amount;
        vaults[token].amount -= amount;
        ...
    }
```

Recommendation

It is recommended to refactor the contract to use two separate variables: `meterDay` and `meterAmount`. Upon each withdrawal attempt, the contract should check if `block.timestamp / 1 days` is different from `meterDay`. If they are different, it implies a new day has started and as such, the `meterAmount` should be reset to zero. This

approach not only makes the logic more straightforward but also potentially optimizes gas usage and improves clarity in daily withdrawal tracking. The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

DSO - Data Structure Optimization

Criticality	Minor / Informative
Location	contracts/Vault.sol#L55
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The contract is currently designed with two separate mappings, `keeper` and `mkeeper`, to manage the status and existence of `keeper` addresses. This design approach is gas-inefficient, leading to potentially higher gas costs for contract interactions. Rather than using two distinct mappings, the contract can be optimized by using a single mapping from the `address` to an `enum`. This `enum` can have three states: `nonExistent`, `Disabled`, and `Enabled`. Such a structure would consolidate the data pertaining to `keeper` addresses into a more efficient and readable format.

```
    function safe(address _keeper) external {
        require(msg.sender == Guard, "Vault: not guard");
        require(mkeeper[_keeper], "Vault: not keeper");
        keeper[_keeper] = !keeper[_keeper];
    }
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

It is recommended to refactor the contract to utilize a single mapping that maps keeper addresses to an enum, which can represent the three states - `nonExistent`, `Disabled`, and `Enabled`.

CR - Centralization Risk

Criticality	Minor / Informative
Location	contracts/Vault.sol#L66,36
Status	Unresolved

Description

The contract uses specific roles, namely `Guard` and `keeper`, which play pivotal roles in the operation and management of the contract. The `Guard` has the authority to update the state of the `keeper` addresses, which means adding or removing them. More critically, the `Guard` has the capability to include its address as a `keeper` address. The `keeper` role is crucial, as these users are responsible for depositing and withdrawing tokens from the contract. This implementation introduces a centralization risk, where a single address, the `Guard,` or the `keeper` has substantial power over the contract, potentially compromising its decentralized nature and leading to central points of failure or abuse.

```
    function updateKeeper(address _keeper, bool state) external
{
        require(msg.sender == Guard, "Vault: not guard");
        require(Update, "Vault: not add");
        keeper[_keeper] = state;
        mkeeper[_keeper] = state;
    }

    function withdraw(address token, address to, uint256
amount) external {
        require(keeper[msg.sender], "Vault: not keeper");
        ...
    }
```

Recommendation

It Is recommended to implement a more decentralized governance mechanism for role management. The team should meticulously manage the private keys associated with the `Guard` and `keeper` addresses' wallets. Additionally, the `Guard` should exercise caution when adding new `keeper` addresses to the system. We strongly recommend a powerful

security mechanism that will prevent a single user from accessing the contract admin functions. Some suggestions are:

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-sign wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.
- Renouncing the ownership will eliminate the threats but it is non-reversible.

MC - Missing Check

Criticality	Minor / Informative
Location	contracts/Vault.sol#L61
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

Specifically, the contract contains the `setLimit` function which allows the `Guard` to set the `Limit` value. However there is no sanity check to ensure that the `Limit` value stays within a specified range. Specifically, the function permits the Limit to be set to any arbitrary value, which could be exceedingly large (greater than 10,000).

```
    function setLimit(uint256 limit) external {
        require(msg.sender == Guard, "Vault: not guard");
        Limit = limit;
    }
```

Recommendation

The team is advised to properly check the variables according to the required specifications. It is recommended to introduce additional checks within the `setLimit` function to ensure that the value of Limit does not exceed `10000` and does not fall below zero. This would provide a safety mechanism against inadvertent attempts to set the `Limit` value to a number outside the acceptable range, ensuring the system operates within the expected parameters. Implementing these checks would enhance the robustness and predictability of the contract's behavior.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	contracts/Vault.sol#L56,62,67,74
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(msg.sender == Guard, "Vault: not guard");
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	contracts/Vault.sol#L66
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The contract updates the `keeper` and `mkeeper` status of an address even if its current state is the same as the one passed as an argument. As a result, the contract performs redundant storage writes.

```
    function updateKeeper(address _keeper, bool state) external
{
        require(msg.sender == Guard, "Vault: not guard");
        require(Update, "Vault: not add");
        keeper[_keeper] = state;
        mkeeper[_keeper] = state;
    }
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	contracts/Vault.sol#L28,36,61
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
   function deposit(address token, address stable, uint256
amount) external {
        require(keeper[msg.sender], "Vault: not keeper");
        if (vaults[token].stable == address(0)) {
            vaults[token].stable = stable;
        }
        vaults[token].amount += amount;
    }

    function withdraw(address token, address to, uint256
amount) external {
        require(keeper[msg.sender], "Vault: not keeper");
        require(vaults[token].amount >= amount, "Vault: not
enough");
        require(vaults[token].stable != address(0), "Vault: not
exist");
        uint256 ds = block.timestamp / 1 days;
        require(
            meter[ds] + amount <=
                ((meter[ds] + vaults[token].amount) * Limit) /
10000,
            "Vault: limit"
        );
        meter[ds] += amount;
        vaults[token].amount -= amount;
        SafeERC20.safeTransfer(IERC20(vaults[token].stable),
to, amount);
    }

    function setLimit(uint256 limit) external {
        require(msg.sender == Guard, "Vault: not guard");
        Limit = limit;
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	contracts/Vault.sol#L21
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
Guard
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/Vault.sol#L9,10,11,55,66@openzeppelin/contracts/token/ERC20/extensions/draft-IERC20Permit.sol#L59
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 public Limit = 3000
address public Guard
bool public Update
address _keeper
function DOMAIN_SEPARATOR() external view returns (bytes32);
```

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.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	contracts/Vault.sol#L63
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
Limit = limit
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	@openzeppelin/contracts/utils/Address.sol#L60,85,114,145,155,170,180,219@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#L30,46,61,70,83
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 sendValue(address payable recipient, uint256 amount)
internal {
        require(address(this).balance >= amount, "Address:
insufficient balance");

        (bool success, ) = recipient.call{value: amount}("");
        require(success, "Address: unable to send value,
recipient may have reverted");
    }

function functionCall(address target, bytes memory data)
internal returns (bytes memory) {
        return functionCallWithValue(target, data, 0, "Address:
low-level call failed");
    }

...
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/Vault.sol#L21
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
Guard = guard
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	@openzeppelin/contracts/utils/Address.sol#L236
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {
                let returndata_size := mload(returndata)
                revert(add(32, returndata), returndata_size)
            }
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	contracts/Vault.sol#L2@openzeppelin/contracts/utils/Address.sol#L4@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#L4@openzeppelin/contracts/token/ERC20/IERC20.sol#L4@openzeppelin/contracts/token/ERC20/extensions/draft-IERC20Permit.sol#L4
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity ^0.8.0;
pragma solidity ^0.8.1;
pragma solidity ^0.8.9;
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	contracts/Vault.sol#L2@openzeppelin/contracts/utils/Address.sol#L4@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#L4@openzeppelin/contracts/token/ERC20/IERC20.sol#L4@openzeppelin/contracts/token/ERC20/extensions/draft-IERC20Permit.sol#L4
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 ^0.8.9;
pragma solidity ^0.8.1;
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.

Functions Analysis

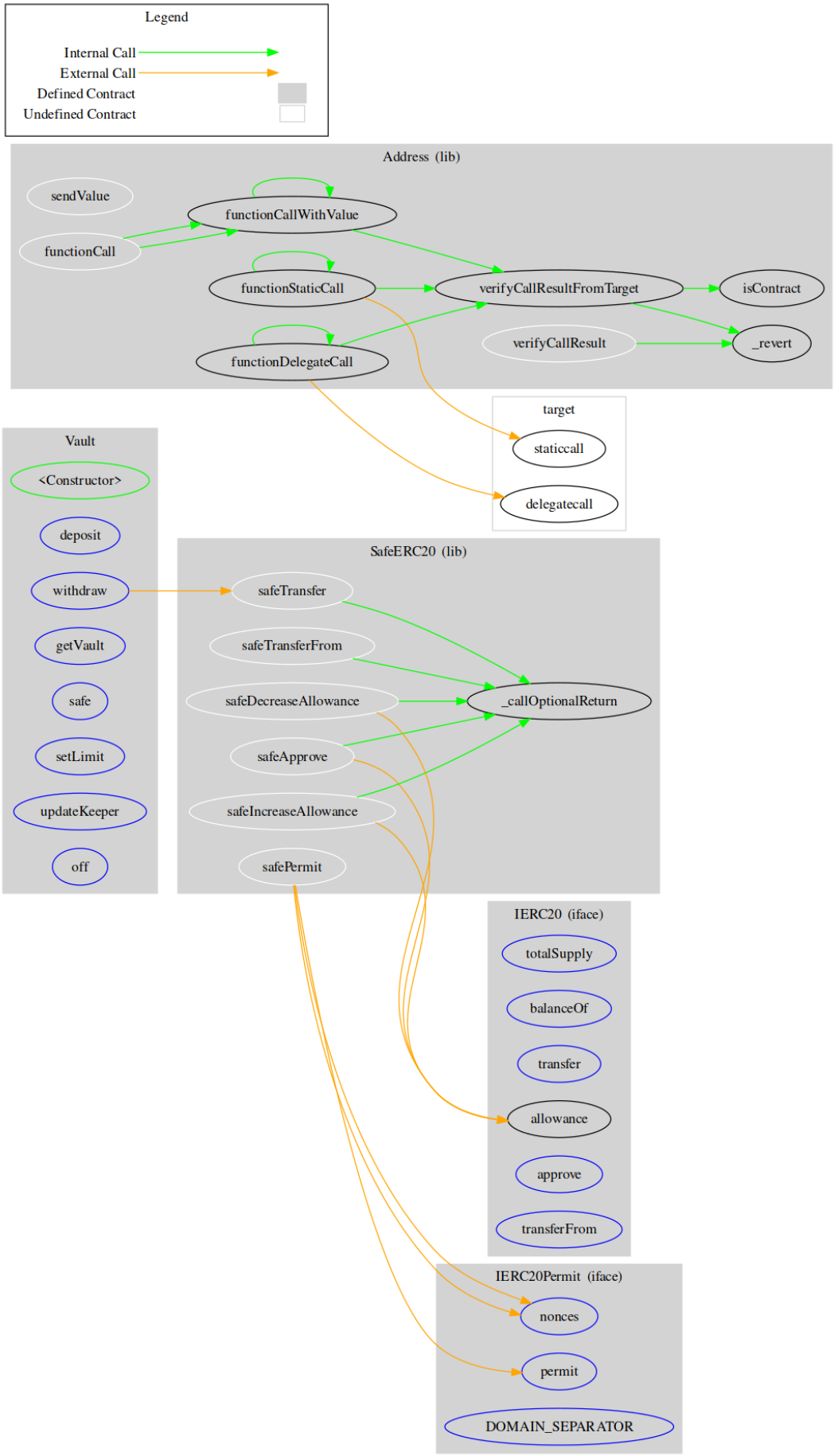
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Vault	Implementation			
		Public	✓	-
	deposit	External	✓	-
	withdraw	External	✓	-
	getVault	External		-
	safe	External	✓	-
	setLimit	External	✓	-
	updateKeeper	External	✓	-
	off	External	✓	-
Address	Library			
	isContract	Internal		
	sendValue	Internal	✓	
	functionCall	Internal	✓	
	functionCall	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionStaticCall	Internal		

	functionStaticCall	Internal		
	functionDelegateCall	Internal	✓	
	functionDelegateCall	Internal	✓	
	verifyCallResultFromTarget	Internal		
	verifyCallResult	Internal		
	_revert	Private		
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
SafeERC20	Library			
	safeTransfer	Internal	✓	
	safeTransferFrom	Internal	✓	
	safeApprove	Internal	✓	
	safeIncreaseAllowance	Internal	✓	
	safeDecreaseAllowance	Internal	✓	
	safePermit	Internal	✓	
	_callOptionalReturn	Private	✓	

IERC20Permit	Interface			
	permit	External	✓	-
	nonces	External		-
	DOMAIN_SEPARATOR	External		-

Inheritance Graph

Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Address (lib)
sendValue
functionCallWithValue
functionCall
functionStaticCall
verifyCallResultFromTarget
isContract
verifyCallResult
_revert
functionDelegateCall
target
staticcall
delegatecall
Vault
<Constructor>
deposit
withdraw
getVault
safe
setLimit
updateKeeper
off
SafeERC20 (lib)
safeTransfer
safeTransferFrom
safeDecreaseAllowance
_callOptionalReturn
safeApprove
safeIncreaseAllowance
safePermit
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
IERC20Permit (iface)
nonces
permit
DOMAIN_SEPARATOR

Summary

RandFi contract implements a locker mechanism. The Vault contract is designed to manage, secure, and regulate the deposit and withdrawal of tokens while ensuring that only specific roles have the permission to carry out particular actions. This audit investigates security issues, business logic concerns, and potential improvements. RandFi is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler errors or critical issues. The contract Owner can access some admin functions that can not be used in a malicious way to disturb the users' transactions.

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