GoodDollar

CompoundStakingV2

Smart Contract Audit Report

May 30, 2022

Introduction

1. About GoodDollar

GoodDollar is a 100% non-profit foundation looking to secure financial freedom for every single person in the world by launching a digital coin, built on the blockchain and based on the principles of universal basic income (UBI). GoodDollar: Changing the Balance, For Good.

About Contract:
GoodDollar project is launching publicly. Its mechanism allows people & organizations to lock funds into an interest-bearing decentralized protocol, currently compound.finance, and donate its created interest towards the Global Basic Income cause. Global Basic Income can be claimed by anyone who proves they are not a bot.

Visit https://www.gooddollar.org/ to know more about it.

2. About ImmuneBytes

ImmuneBytes is a security start-up to provides professional services in the blockchain space. The team has hands-on experience in conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and have a great understanding of DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, and dydx.

The team has been able to secure 105+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-ups with a detailed analysis of the system ensuring security and managing the overall project.

Visit http://immunebytes.com/ to know more about the services.

Documentation Details

The GoodDollar team has provided the following doc for the purpose of audit:

1. GoodDollar High Level Overview.pdf

Audit Process & Methodology

ImmuneBytes team has performed thorough testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line-by-line inspection of the Smart Contract in order to find any potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the Unit testing phase, we run unit tests written by the developer in order to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors which includes -

1. Testing the functionality of the Smart Contract to determine proper logic has been followed throughout.
2. Analyzing the complexity of the code by thorough, manual review of the code, line-by-line.
3. Deploying the code on testnet using multiple clients to run live tests.
4. Analyzing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
5. Checking whether all the libraries used in the code are on the latest version.
6. Analyzing the security of the on-chain data.

Audit Details

- Project Name: GoodDollar
- Contracts Name: CompoundStakingV2.sol, SimpleStakingV2
- Languages: Solidity(Smart contract), Typescript (Unit Testing)
- Github commits for the audit: 7ee04d23fb8ad3468a041e3f907d9310fb5ffa1d
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Contract Library, Slither, SmartCheck

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 into 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
    a. Correctness
    b. Readability
    c. Sections of code with high complexity
    d. Quantity and quality of test coverage

Security Level Reference

Every issue in this report were assigned a severity level from the following:

High severity issues will bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Issues	High	Medium	Low
Open	-	-	-
Closed	-	1	5

This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

CompoundStakingV2.sol

High Severity Issues

No issues were found.

Medium Severity Issues

1. **Multiplication is being performed on the result of Division**

Explanation:
During the automated testing of the **GoodCompoundStakingV2** contract, it was found that 2 functions in the contract are performing multiplication on the result of a Division.

```
269    function iTokenWorthInToken(uint256 _amount)
270        internal
271        view
272        override
273        returns (uint256)
274    {
275        uint256 er = cERC20(address(iToken)).exchangeRateStored();
276        (uint256 decimalDifference, bool caseType) = tokenDecimalPrecision();
277        uint256 mantissa = 18 + tokenDecimal() - iTokenDecimal();
278        uint256 tokenWorth = caseType == true
279            ? (_amount * (10**decimalDifference) * er) / 10**mantissa
280            : ((_amount / (10**decimalDifference)) * er) / 10**mantissa; // calculation
```

Integer Divisions in Solidity might truncate. Moreover, this performing division before multiplication might lead to a loss of precision.

The following functions involve division before multiplication in the mentioned lines:

- **iTokenWorthInToken** at 278-280
- **tokenWorthIniToken** at 297-299

Recommendation:
Solidity doesn't encourage arithmetic operations that involve division before multiplication. Therefore the above-mentioned function should be checked once and redesigned if they do not lead to the expected results.

Low Severity Issues

1. **Absence of input validation in the setcollectInterestGasCostParams() function**
   Line no -307-314

   Description:
   The setcollectInterestGasCostParams() function doesn't include any input validation on the uint32 arguments being passed to the function.

   Although the function is only accessible by the owner(avatar), collectInterestGasCost and compCollectGasCost are imperative state variables as these are being used for gas costs during interest transfers).

   Recommendation:
   Input validations must be included before updating important state variables

2. **No Events emitted after imperative State Variable modification**
   Line no -307-314

   Description:
   Functions that update an imperative arithmetic state variable contract should emit an event after the state modification.

   The setcollectInterestGasCostParams() function modifies some crucial arithmetic parameters like collectInterestGasCost, compCollectGasCost in the GoodCompoundStakingV2 contract but doesn't emit any event after the updation.

   Since there is no event emitted on updating these variables, it might be difficult to track it off-chain.

   Recommendation:
   An event should be fired after changing crucial arithmetic state variables.

3. **Zero Address validations not found in initializer function**

   Description:
   The init() function of GoodCompoundStakingV2 doesn't include adequate zero address validations for the addresses passed to the function.

   Recommendation:
   A require statement should be included in such functions to ensure no zero address is passed in the arguments.

Recommendation / Informational

1. **Redundant comparisons to boolean Constants**
   Line no: 278, 297

   Description:
   Boolean constants can directly be used in conditional statements or require statements.

```
    ftrace | funcSig
289 function tokenWorthIniToken(uint256 _amount) //@audit-ok
290     public
291     view
292     returns (uint256 tokenWorth)
293 {
294     uint256 er = cERC20(address(iToken)).exchangeRateStored();
295     (uint256 decimalDifference, bool caseType) = tokenDecimalPrecision();
296     uint256 mantissa = 18 + tokenDecimal() - iTokenDecimal();
297     tokenWorth = caseType == true
```

   Therefore, it's not considered a better practice to explicitly use TRUE or FALSE in the require statements.

   Recommendation:
   The equality to boolean constants could be removed from the above-mentioned line.

2. **Unlocked Pragma statements found in the contracts**
   Line no: 2

   Explanation:
   During the code review, it was found that the contracts included unlocked pragma solidity version statements.

It’s not considered a better practice in Smart contract development to do so as it might lead to accidental deployment to a version with unfixed bugs.

Recommendation:
It’s always recommended to lock pragma statements to a specific version while writing contracts.

3. **Coding Style Issues in the Contract**

Explanation:
Code readability of a Smart Contract is largely influenced by the Coding Style issues and in some specific scenarios may lead to bugs in the future.

During the automated testing, it was found that the GoodCompoundStakingV2 contract had quite a few code style issues.

Recommendation:
Therefore, it is recommended to fix the issues like naming convention, indentation, and code layout issues in a smart contract.

CompoundStakingV2.sol

High Severity Issues

No issues were found.

Medium Severity Issues

No issues were found.

Low Severity Issues

1. **recover() function doesn't adequately ensure the withdrawable amount of tokens**
   Line no: 435 - 444

   Description:
   The recover() function allows the owner of the contract to pass any erc20 token address to recover the withdrawable token for the given address.

```
435     function recover(ERC20 _token) public {
436         _onlyAvatar();
437         uint256 toWithdraw = _token.balanceOf(address(this));
438 
439         // recover left iToken(stakers token) only when all stakes have been withdrawn
440         if (address(_token) == address(iToken)) {
441             require(totalProductivity == 0 && isPaused, "recover");
442         }
```

   However, the function doesn't include any adequate check to ensure that the total withdrawable amount that is stored in the local variable toWithdraw, is actually more than zero.

   This leads to a scenario where there could be no token balance for a given address but the function would still execute since it lacks adequate validations and lead to loss of gas.

   Recommendation:
   The function should include adequate checks to ensure that the withdrawable token amount is actually more than zero.

2. **No Events are emitted after updating the maxLiquidityPercentageSwap variable**
   Line no -307-314

   Description:
   The setMaxLiquidityPercentageSwap() function updates the maxLiquidityPercentageSwap state variable but doesn't emit any event after the modification.

   Since there is no event emitted on updating these variables, it might be difficult to track it off-chain.

   Recommendation:
   An event should be fired after changing crucial arithmetic state variables.

Recommendation / Informational

1. **Redundant comparisons to boolean constants can be avoided**
   Line no: 190

   Description:
   Boolean constants can directly be used in conditional statements or require statements.

   Therefore, it's not considered a better practice to explicitly use TRUE or FALSE in the require statements. The require statement at Line 190 involves an unnecessary comparison to the boolean constant which can be avoided.

   Recommendation:
   The equality to boolean constants could be removed from the above-mentioned line.

2. **Unlocked Pragma statements found in the contracts**
   Line no: 2

   Explanation:
   During the code review, it was found that the contracts included unlocked pragma solidity version statements.

   It’s not considered a better practice in Smart contract development to do so as it might lead to accidental deployment to a version with unfixed bugs.

   Recommendation:
   It’s always recommended to lock pragma statements to a specific version while writing contracts.

Automated Audit Result

```
Compiled with solc
Number of lines: 9018 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 70 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 90
Number of informational issues: 624
Number of low issues: 47
Number of medium issues: 230
Number of high issues: 18
ERCs: ERC165, ERC20
```

Name	# functions	ERCS	ERC20 info	Complex code	Features
Avatar	3			No	
Controller	11			No	
ReputationInterface	7			No	
SchemeRegistrar	1			No	
IntVoteInterface	8			No	
cERC20	16	ERC20	∞ Minting Approve Race Cond.	No	AbiEncoderV2
IGoodDollar	18	ERC20	∞ Minting Approve Race Cond.	No	AbiEncoderV2
IERC2917	18	ERC20	∞ Minting Approve Race Cond.	No	AbiEncoderV2
Staking	3			No	AbiEncoderV2
Uniswap	9			No	Receive ETH AbiEncoderV2
UniswapFactory	1			No	AbiEncoderV2
UniswapPair	6			No	AbiEncoderV2
Reserve	1			No	AbiEncoderV2
IIdentity	7			No	AbiEncoderV2
IUBIScheme	3			No	AbiEncoderV2
IFirstClaimPool	2			No	AbiEncoderV2
ProxyAdmin	5			No	AbiEncoderV2

					Proxy
AggregatorV3Interface	5			No	AbiEncoderV2
ILendingPool	3			No	AbiEncoderV2
IDonationStaking	1			No	Receive ETH
					AbiEncoderV2
INameService	1			No	AbiEncoderV2
IAaveIncentivesController	2			No	AbiEncoderV2
IGoodStaking	6			No	AbiEncoderV2
IAdminWallet	4			No	AbiEncoderV2
GReputation	92	ERC165		Yes	Receive ETH
					Ecrecover
					Delegatecall
					Assembly
					Upgradeable
StakersDistribution	49			Yes	Receive ETH
					Delegatecall
					Tokens interaction
					Upgradeable
GoodMarketMaker	46			No	Receive ETH
					Delegatecall
					Tokens interaction
					Upgradeable
ContributionCalc	2			No	
GoodReserveCDai	131	ERC20,ERC165	Pausable	No	Receive ETH
			∞ Minting		Delegatecall
			Approve Race Cond.		Tokens interaction
					Upgradeable
GoodFundManager	42			Yes	Receive ETH
					Delegatecall
					Tokens interaction
					Upgradeable
UniswapV2SwapHelper	2			No	Send ETH
GoodCompoundStakingV2	88	ERC20	Pausable	Yes	Send ETH
			No Minting		Tokens interaction
			Approve Race Cond.		Upgradeable
BancorFormula	39			Yes	
DataTypes	0			No	
NameService	27			No	Receive ETH

					Delegatecall
					Upgradeable
IBeaconUpgradeable	1			No	Upgradeable
AddressUpgradeable	9			No	Send ETH
					Assembly
					Upgradeable
StorageSlotUpgradeable	4			No	Assembly
					Upgradeable
StringsUpgradeable	4			Yes	Upgradeable
MerkleProofUpgradeable	1			No	Upgradeable
SafeMathUpgradeable	13			No	Upgradeable
EnumerableSetUpgradeable	24			No	Assembly
					Upgradeable

contracts/staking/compound/GoodCompoundStakingV2.sol analyzed (70 contracts)

Concluding Remarks

While conducting the audits of the GoodDollar smart contracts, it was observed that the contracts contain Medium and Low severity issues.

Our auditors suggest that Medium and Low severity issues should be resolved by the developers. The recommendations given will improve the operations of the smart contract.

Disclaimer

ImmuneBytes's audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Our team does not endorse the GoodDollar platform or its product nor this audit is investment advice.
Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for the code refactor by the team on critical issues.

ImmuneBytes