Smart Contract Audit Report for GoodDollar

Testers

1. Or Duan
2. Avigdor Sason Cohen
3. Ido Shdaimah

Table of Contents

Management Summary

Good Dollar contacted Sayfer to perform a security audit on their smart contracts.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for GoodDollar's smart contracts.

Over the research period of 40 research hours, we discovered 7 vulnerabilities in the contract.

After a review by the Sayfer team, we validate that the system was patched correctly.

Risk Methodology

At Sayfer, we are committed to delivering the highest quality smart contract audits to our clients. That's why we have implemented a comprehensive risk assessment model to evaluate the severity of our findings and provide our clients with the best possible recommendations for mitigation.

Our risk assessment model is based on two key factors: **IMPACT** and **LIKELIHOOD**. Impact refers to the potential harm that could result from an issue, such as financial loss, reputational damage, or a non-operational system. Likelihood refers to the probability that an issue will occur, taking into account factors such as the complexity of the contract and the number of potential attackers.

By combining these two factors, we can create a comprehensive understanding of the risk posed by a particular issue and provide our clients with a clear and actionable assessment of the severity of the issue. This approach allows us to prioritize our recommendations and ensure that our clients receive the best possible advice on how to protect their smart contracts.

Risk is defined as follows:

Overall Risk Security

IMPACT		LIKELIHOOD		
HIGH		Medium	High	Critical
MEDIUM		Low	Medium	High
LOW		Informational	Low	Medium
		LOW	MEDIUM	HIGH

Vulnerabilities by Risk
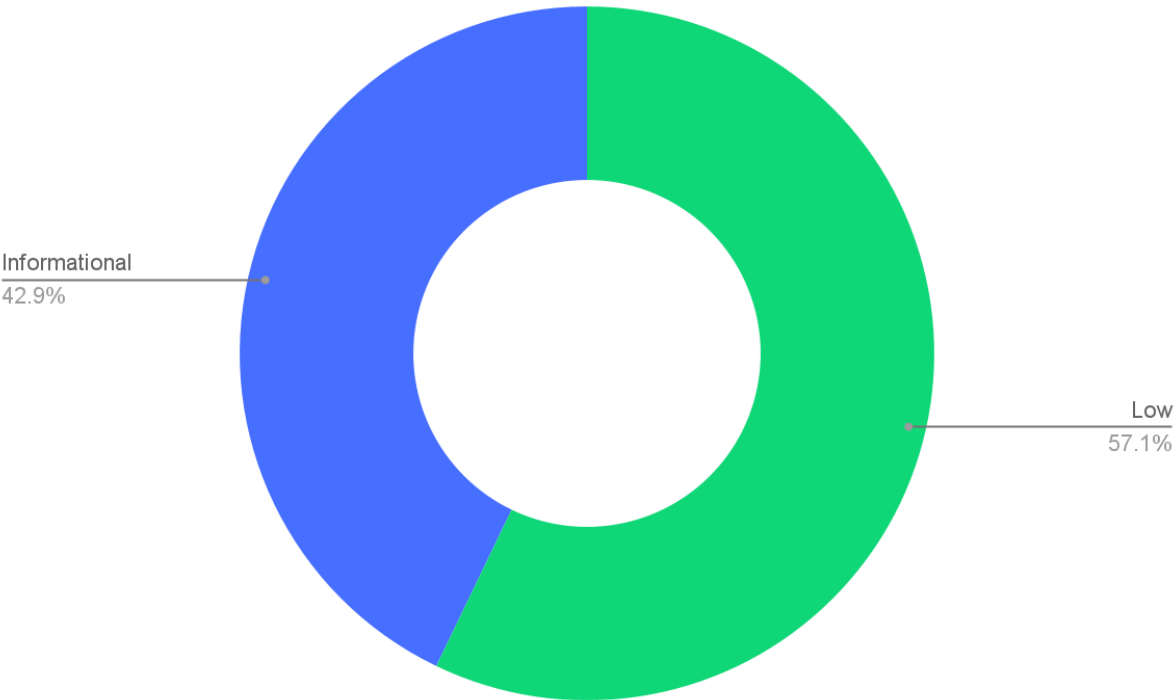
Risk	Low	Medium	High	Critical	Informational
# of issues	4	0	0	0	3

Approach

Introduction

GoodDollar contacted Sayfer to perform a security audit on their smart contracts.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for the aforementioned contracts.

Scope Overview

Together with the client team we defined the following contract as the scope of the project. Commit hashes:

- **GoodProtocol/reserve-single-distribution:**
  d201eded0f2cadb74fd26817ffc84b38afebbf73

Contract/Scripts	Sha-256
GoodReserveCDai.sol	054e7f5140aa56d93287feed3adf614523e35b86e6ee39e3ab8060a6c22e396f
GoodFundManager.sol	6fd055243e733ee9c1367e4b045420be9221c7e892fa5eb2eaa7fe9e430bef5f
MultichainFeeFormula.sol	215d9b962fa8bc2265cd19cdac762344918812b953fa78e27b5330d1a8862832
multichain-formula-upgrade.ts	0ddf7a4959ba119515e4d45f72e1357a24dd65cc17abfcead9fab7ef35b94c95
reserve-upgrade.ts	935a34164555ef1466ca1908100c9df34573ea0cdf43f3bcef6904e911876cfb

- **GoodBridge/mbp-bridge:**
  2a3f7c36fb09755a1599d52a8d853ca355ef82bf

Contract/Scripts	Sha-256
AxelarHandlerUpgradeable.sol	a8d7afc6ab61b4fac07def6d45ccc1da4b5ccd7531754cb054c81662a98e9291
BridgeHelperLibrary.sol	249a00467c4bc36ea0ea557b880a25abbd1593c4e4cf20ab1915cc282ab7df41
IMessagePassingBridge.sol	5022cc139e01b20e2ae159bb9c2927b6d6865d1d4bf1a151a0e3534c234f3060
LZHandlerUpgradeable.sol	31e48e357f553648eb72924607e5a68c0db3003efb133df60608d81997c33625
MessagePassingBridge.sol	ac66f603527656f6cf66e0b0b44fce6b96e729afda85f87b3c19580b3950ce64

Our tests were performed in August 2023

Scope Validation

We began by ensuring that the scope defined to us by the client was technically logical. Deciding what scope is right for a given system is part of the initial discussion.

Threat Model

We defined that the largest current threat to the system is the ability of malicious users to steal funds from the contract.

SAYFER

Protocol Overview

Protocol Introduction

The GoodDollar protocol is a community-driven, distributed framework designed to generate, fund, and distribute global basic income via the GoodDollar token (hereafter "G$"). G$ is an ERC-20 digital asset built on the Ethereum blockchain that operates within the emerging ecosystem of decentralized and open finance. GoodDollar leverages new protocols and smart contracts across the ecosystem to deliver its basic income economy.

Security Evaluation

The following test cases were the guideline while auditing the system. This checklist is a modified version of the SCSVS v1.2, with improved grammar, clarity, conciseness, and additional criteria. Where there is a gap in the numbering, an original criterion was removed. Criteria that are marked with an asterisk were added by us.

Architecture, Design and Threat Modeling	Test Name
G1.2	Every introduced design change is preceded by threat modeling.
G1.3	The documentation clearly and precisely defines all trust boundaries in the contract (trusted relations with other contracts and significant data flows).
G1.4	The SCSVS, security requirements or policy is available to all developers and testers.
G1.5	The events for the (state changing/crucial for business) operations are defined.
G1.6	The project includes a mechanism that can temporarily stop sensitive functionalities in case of an attack. This mechanism should not block users' access to their assets (e.g. tokens).
G1.7	The amount of unused cryptocurrencies kept on the contract is controlled and at the minimum acceptable level so as not to become a potential target of an attack.
G1.8	If the fallback function can be called by anyone, it is included in the threat model.
G1.9	Business logic is consistent. Important changes in the logic should be applied in all contracts.
G1.10	Automatic code analysis tools are employed to detect vulnerabilities.
G1.11	The latest major release of Solidity is used.
G1.12	When using an external implementation of a contract, the most recent version is used.
G1.13	When functions are overridden to extend functionality, the super keyword is used to maintain previous functionality.
G1.14	The order of inheritance is carefully specified.
G1.15	There is a component that monitors contract activity using events.
G1.16	The threat model includes whale transactions.
G1.17	The leakage of one private key does not compromise the security of the entire project.

Policies and Procedures	Test Name

G2.2	The system's security is under constant monitoring (e.g. the expected level of funds).
G2.3	There is a policy to track new security vulnerabilities and to update libraries to the latest secure version.
G2.4	The security department can be publicly contacted and that the procedure for handling reported bugs (e.g., thorough bug bounty) is well-defined.
G2.5	The process of adding new components to the system is well defined.
G2.6	The process of major system changes involves threat modeling by an external company.
G2.7	The process of adding and updating components to the system includes a security audit by an external company.
G2.8	In the event of a hack, there's a clear and well known mitigation procedure in place.
G2.9	The procedure in the event of a hack clearly defines which persons are to execute the required actions.
G2.10	The procedure includes alarming other projects about the hack through trusted channels.
G2.11	A private key leak mitigation procedure is defined.

Upgradability	Test Name
G2.2	Before upgrading, an emulation is made in a fork of the main network and everything works as expected on the local copy.
G2.3	The upgrade process is executed by a multisig contract where more than one person must approve the operation.
G2.4	Timelocks are used for important operations so that the users have time to observe upcoming changes (please note that removing potential vulnerabilities in this case may be more difficult).
G2.5	*initialize()* can only be called once.
G2.6	*initialize()* can only be called by an authorized role through appropriate modifiers (e.g. *initializer*, *onlyOwner*).
G2.7	The update process is done in a single transaction so that no one can front-run it.
G2.8	Upgradeable contracts have reserved gap on slots to prevent overwriting.
G2.9	The number of reserved (as a gap) slots has been reduced appropriately if new variables have been added.
G2.10	There are no changes in the order in which the contract state variables are declared, nor their types.
G2.11	New values returned by the functions are the same as in previous versions of the contract (e.g. *owner()*, *balanceOf(address)*).
G2.12	The implementation is initialized.
G2.13	The implementation can't be destroyed.

Business Logic	Test Name
G4.2	The contract logic and protocol parameters implementation corresponds to the documentation.
G4.3	The business logic proceeds in a sequential step order and it is not possible to skip steps or to do it in a different order than designed.
G4.4	The contract has correctly enforced business limits.
G4.5	The business logic does not rely on the values retrieved from untrusted contracts (especially when there are multiple calls to the same contract in a single flow).
G4.6	The business logic does not rely on the contract's balance (e.g., *balance == 0*).
G4.7	Sensitive operations do not depend on block data (e.g., *block hash*, *timestamp*).
G4.8	The contract uses mechanisms that mitigate transaction-ordering (front-running) attacks (e.g. pre-commit schemes).
G4.9	The contract does not send funds automatically, but lets users withdraw funds in separate transactions instead.

Access Control	Test Name
G5.2	The principle of the least privilege is upheld. Other contracts should only be able to access functions and data for which they possess specific authorization.
G5.3	New contracts with access to the audited contract adhere to the principle of minimum rights by default. Contracts should have a minimal or no permissions until access to the new features is explicitly granted.
G5.4	The creator of the contract complies with the principle of the least privilege and their rights strictly follow those outlined in the documentation.
G5.5	The contract enforces the access control rules specified in a trusted contract, especially if the dApp client-side access control is present and could be bypassed.
G5.6	Calls to external contracts are only allowed if necessary.
G5.7	Modifier code is clear and simple. The logic should not contain external calls to untrusted contracts.
G5.8	All user and data attributes used by access controls are kept in trusted contracts and cannot be manipulated by other contracts unless specifically authorized.
G5.9	the access controls fail securely, including when a revert occurs.
G5.10	If the input (function parameters) is validated, the positive validation approach (whitelisting) is used where possible.

Communication	Test Name
G6.2	Libraries that are not part of the application (but the smart contract relies on to operate) are identified.

G6.3	Delegate call is not used with untrusted contracts.
G6.4	Third party contracts do not shadow special functions (e.g. revert).
G6.5	The contract does not check whether the address is a contract using *extcodesize* opcode.
G6.6	Re-entrancy attacks are mitigated by blocking recursive calls from other contracts and following the Check-Effects-Interactions pattern. Do not use the *send* function unless it is a must.
G6.7	The result of low-level function calls (e.g. *send*, *delegatecall*, *call*) from other contracts is checked.
G6.8	Contract relies on the data provided by the right sender and does not rely on tx.origin value.

Arithmetic	Test Name
G7.2	The values and math operations are resistant to integer overflows. Use SafeMath library for arithmetic operations before solidity 0.8.*.
G7.3	the unchecked code snippets from Solidity ≥ 0.8.* do not introduce integer under/overflows.
G7.4	Extreme values (e.g. maximum and minimum values of the variable type) are considered and do not change the logic flow of the contract.
G7.5	Non-strict inequality is used for balance equality.
G7.6	Correct orders of magnitude are used in the calculations.
G7.7	In calculations, multiplication is performed before division for accuracy.
G7.8	The contract does not assume fixed-point precision and uses a multiplier or store both the numerator and denominator.

Denial of Service	Test Name
G8.2	The contract does not iterate over unbound loops.
G8.3	Self-destruct functionality is used only if necessary. If it is included in the contract, it should be clearly described in the documentation.
G8.4	The business logic isn't blocked if an actor (e.g. contract, account, oracle) is absent.
G8.5	The business logic does not disincentivize users to use contracts (e.g. the cost of transaction is higher than the profit).
G8.6	Expressions of functions assert or require have a passing variant.
G8.7	If the fallback function is not callable by anyone, it is not blocking contract functionalities.
G8.8	There are no costly operations in a loop.
G8.9	There are no calls to untrusted contracts in a loop.
G8.10	If there is a possibility of suspending the operation of the contract, it is also

	possible to resume it.
G8.11	If whitelists and blacklists are used, they do not interfere with normal operation of the system.
G8.12	There is no DoS caused by overflows and underflows.

Blockchain Data	Test Name
G9.2	Any saved data in contracts is not considered secure or private (even private variables).
G9.3	No confidential data is stored in the blockchain (passwords, personal data, token etc.).
G9.4	Contracts do not use string literals as keys for mappings. Global constants are used instead to prevent Homoglyph attack.
G9.5	Contract does not trivially generate pseudorandom numbers based on the information from blockchain (e.g. seeding with the block number).

Gas Usage and Limitations	Test Name
G10.2	Gas usage is anticipated, defined and has clear limitations that cannot be exceeded. Both code structure and malicious input should not cause gas exhaustion.
G10.3	Function execution and functionality does not depend on hard-coded gas fees (they are bound to vary).

Clarity and Readability	Test Name
G11.2	The logic is clear and modularized in multiple simple contracts and functions.
G11.3	Each contract has a short 1-2 sentence comment that explains its purpose and functionality.
G11.4	Off-the-shelf implementations are used, this is made clear in comment. If these implementations have been modified, the modifications are noted throughout the contract.
G11.5	The inheritance order is taken into account in contracts that use multiple inheritance and shadow functions.
G11.6	Where possible, contracts use existing tested code (e.g. token contracts or mechanisms like *ownable*) instead of implementing their own.
G11.7	Consistent naming patterns are followed throughout the project.
G11.8	Variables have distinctive names.
G11.9	All storage variables are initialized.
G11.10	Functions with specified return type return a value of that type.

G11.11	All functions and variables are used.
G11.12	*require* is used instead of *revert* in *if* statements.
G11.13	The *assert* function is used to test for internal errors and the *require* function is used to ensure a valid condition in input from users and external contracts.
G11.14	Assembly code is only used if necessary.

Test Coverage	Test Name
G12.2	Abuse narratives detailed in the threat model are covered by unit tests.
G12.3	Sensitive functions in verified contracts are covered with tests in the development phase.
G12.4	Implementation of verified contracts has been checked for security vulnerabilities using both static and dynamic analysis.
G12.5	Contract specification has been formally verified.
G12.6	The specification and results of the formal verification is included in the documentation.

Decentralized Finance	Test Name
G14.1	The lender's contract does not assume its balance (used to confirm loan repayment) to be changed only with its own functions.
G14.2	Functions that change lenders' balance and/or lend cryptocurrency are non-re-entrant if the smart contract allows borrowing the main platform's cryptocurrency (e.g. Ethereum). It blocks the attacks that update the borrower's balance during the flash loan execution.
G14.3	Flash loan functions can only call predefined functions on the receiving contract. If it is possible, define a trusted subset of contracts to be called. Usually, the sending (borrowing) contract is the one to be called back.
G14.4	If it includes potentially dangerous operations (e.g. sending back more ETH/tokens than borrowed), the receiver's function that handles borrowed ETH or tokens can be called only by the pool and within a process initiated by the receiving contract's owner or another trusted source (e.g. multisig).
G14.5	Calculations of liquidity pool share are performed with the highest possible precision (e.g. if the contribution is calculated for ETH it should be done with 18 digit precision - for Wei, not Ether). The dividend must be multiplied by the 10 to the power of the number of decimal digits (e.g. dividend * 10^18 / divisor).
G14.6	Rewards cannot be calculated and distributed within the same function call that deposits tokens (it should also be defined as non-re-entrant). This protects from momentary fluctuations in shares.
G14.7	Governance contracts are protected from flash loan attacks. One possible

	mitigation technique is to require the process of depositing governance tokens and proposing a change to be executed in different transactions included in different blocks.
G14.8	When using on-chain oracles, contracts are able to pause operations based on the oracles' result (in case of a compromised oracle).
G14.9	External contracts (even trusted ones) that are allowed to change the attributes of a project contract (e.g. token price) have the following limitations implemented: thresholds for the change (e.g. no more/less than 5%) and a limit of updates (e.g. one update per day).
G14.10	Contract attributes that can be updated by the external contracts (even trusted ones) are monitored (e.g. using events) and an incident response procedure is implemented (e.g. during an ongoing attack).
G14.11	Complex math operations that consist of both multiplication and division operations first perform multiplications and then division.
G14.12	When calculating exchange prices (e.g. ETH to token or vice versa), the numerator and denominator are multiplied by the reserves (see the *getInputPrice* function in the *UniswapExchange* contract).

Audit Findings

GoodProtocol/reserve-single-distribution

Minting May Fail Because of Distribution Helper

ID	SAY-01
Status	Fixed
Risk	Low
Business Impact	Minting may revert if the distribution helper (temporarily) reverts.
Location	- GoodReserveCDai.sol:391; mintUBI(uint256, uint256, ERC20)

Description	Previously, the call distributionHelper.onDistribution(gdUBI); was in a try / catch block with the comment "should not prevent mintUBI from completing". This was removed and a revert in the distribution helper will cause the mint call to fail. The function DistributionHelper.distribute (which is called by onDistribution) may for instance revert when the fees are too high temporarily.
	
	 ● GoodReserveCDai.sol:390-391
	 ``solidity
	_mintGoodDollars(address(distributionHelper), gdUBI, false);
	//mintGoodDollars enforces that only minter can call mintUBI
	distributionHelper.onDistribution(gdUBI);
	``

Mitigation	We recommend restoring the old try / catch block.

Sending Funds to Multichain Still Works

ID	SAY-02
Status	Fixed
Risk	Low
Business Impact	Users can still successfully bridge into the defunct multichain bridge.
Location	- contracts/token/MultichainFeeFormula.sol:21; function getTxFees(uint256, uint256, address)
Description	*getTxFees(uint256, uint256, address)* charges a 100% fee when the sender is the multichain token contract. However, when it is the recipient (e.g., when a user interacts with the [vault](#)), no fee is charged and all the funds are transferred to the contract. ● getTxFees(uint256, uint256, address) <pre><code>function getTxFees(
    uint256 value,
    address sender,
    address /*recipient*/
) public view returns (uint256 fee, bool senderPays) {
    senderPays = false;
    if (sender == address(0xD17652350Cfd2A37bA2f947C910987a3B1A1c60d))
        fee = value;
}</code></pre>
Mitigation	We recommend also charging the fee when the recipient is the multichain token contract.

Commented Code

ID	SAY-03
Status	Fixed
Risk	Informational
Business Impact	Old commented code like this may be confusing or be perceived as unprofessional by readers. It is a purely cosmetic issue.
Location	- GoodReserveCDai.sol:382-387; mintUBI(uint256, uint256, address)
Description	The specified function has a few lines of commented test code.

● `mintUBI(uint256, uint256, address)`

```
// console.log(
//   "nonubi %s, sender: %s, fundManager: %s",
//   nonUbiBps,
//   _msgSender(),
//   nameService.getAddress("FUND_MANAGER")
// );
```

Mitigation	This test code can be safely removed in production environment.

Contract Verification in Upgrade Script

ID	SAY-04
Status	Fixed
Risk	Informational
Business Impact	The *GoodFundManager* and *DistributionHelper* contracts will not be verified.
Location	- `reserve-upgrade.ts:83`
Description	The upgrade script only contains a call to verify the updated *GoodReserveCDai* contract. However, the *GoodFundManager* and *DistributionHelper* contracts are also updated, but the updated version is not automatically verified.
Mitigation	● This can be easily rectified by adding two lines of boilerplate: ```if (isProduction) await verifyContract(newReserveImpl, "GoodReserveCDai", networkName); if (isProduction) await verifyContract(newFundmanagerImpl, "GoodFundManager", networkName); if (isProduction) await verifyContract(newDisthelperImpl, "DistributionHelper", networkName);```

GoodBridge/mpb-bridge

Missing Storage Gaps for Upgradeable Contract

ID	SAY-05
Status	Fixed
Risk	Low
Business Impact	Because of missing storage gaps, it will not be possible to add additional variables to the contract when it is upgraded. This can severely impact the upgradability of the system and may result in situations where certain upgrades cannot be done.
Location	- AxelarHandlerUpgradeable.sol
Description	The contract *AxelarHandlerUpgradeable* is intended to be upgradeable and another upgradeable contract (*MessagePassingBridge*) inherits from it. But the contract does not contain any storage gaps to add additional variables in the future.
Mitigation	● This can be mitigated by adding a storage gap array:

```
abstract contract AxelarHandlerUpgradeable is AxelarExecutable {
    using StringToAddress for string;
    using AddressToString for address;

    // An interface for the gas service contract
    IAxelarGasService public immutable gasService;

    // Gap array
    uint256[49] __gap;
                                    { ... }
}
```

In future iterations of the contract, if variables are added and more slots are taken up, the gap array is reduced.

Hardcoded Deprecated Testnet

ID	SAY-06
Status	Open
Risk	Acknowledged
Business Impact	Unless the code is modified, this function will not work properly on other Testnets, potentially breaking the contract's functionality.
Location	- `MessagePassingBridge.sol:355; _bridgeTo(address, address, uint256, uint256, bool, BridgeService, bytes memory, address)`
Description	The function _bridgeTo has a hard-coded chain ID of 5 (ETH Goerli Testnet). This network is already deprecated and will stop working in January 2024.

- `MessagePassingBridge.sol:355-357`

```solidity
if (_chainId() == 1 || _chainId() == 5) {
    if (nativeToken().transferFrom(from, address(this), amount) ==
false) revert TRANSFER_FROM();
} else nativeToken().burnFrom(from, amount);
```

Mitigation	We recommend making the ID(s) configurable. Deployments on other testnets such as Sepolia will then no longer require code changes and could just use a different configuration.

Unnecessary Assignment

ID	SAY-07
Status	Fixed
Risk	Informational
Business Impact	This redundant assignment may slightly impact gas usage.
Location	- BridgeHelperLibrary.sol:40; normalizeFrom18ToTokenDecimals(uint256, uint8)
Description	When decimals are equal to 18, *normalizeFrom18ToTokenDecimals()* performs an additional assignment of amount to normalized in the else branch. However, because this assignment was already performed in line 33, it is redundant.

● normalizeFrom18ToTokenDecimals(uint256, uint8); Notice how lines 33 and 40 perform the same action.

```solidity
function normalizeFrom18ToTokenDecimals(uint256 amount, uint8
decimals) external pure returns (uint256 normalized) {
    normalized = amount;
    if (decimals < 18) {
        uint256 diff = 18 - decimals;
        normalized = amount / 10 ** diff;
    } else if (decimals > 18) {
        uint256 diff = decimals - 18;
        normalized = amount * 10 ** diff;
    } else normalized = amount;
}
```

Mitigation	The *else* branch in line 40 can just be removed.