

Smart Contract Audit Report for GoodDollar

Testers

1. Or Duan
2. Avigdor Sason Cohen

Table of Contents

Table of Contents **2**
Management Summary **3**
Risk Methodology **4**
Vulnerabilities by Risk **5**
Approach **6**
Introduction 6
Scope Overview 6
Scope Validation 6
Threat Model 6
Protocol Overview **7**
Protocol Introduction 7
Security Evaluation **8**
Security Assessment Findings **15**
Absence of Rate Limiting on Outbound Bridge Operations 15
Incorrect Comparison 16
Unchecked ERC20 Transfer Return Value 17
Global Trading Limit Uses Symmetric Bounds unlike Directional Limits 18
whitelistedCount Isn't Decremented When Users Are Blacklisted 20
Bridge Messages from Untrusted Sources Are Not Marked as Executed 21
Bridge Fee Accepts Values Exceeding 100% 23
External Call Execution without Reentrancy Protection 25

Management Summary

GoodDollar contacted Sayfer to perform a security audit on their smart contracts in December 2025.

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 2 weeks, we discovered 8 vulnerabilities in the contract.

Several fixes should be implemented following the report, to ensure the system's security posture is competent.

After a review by the Sayfer team, we certify that all the security issues mentioned in this report have been addressed by the GoodDollar team.

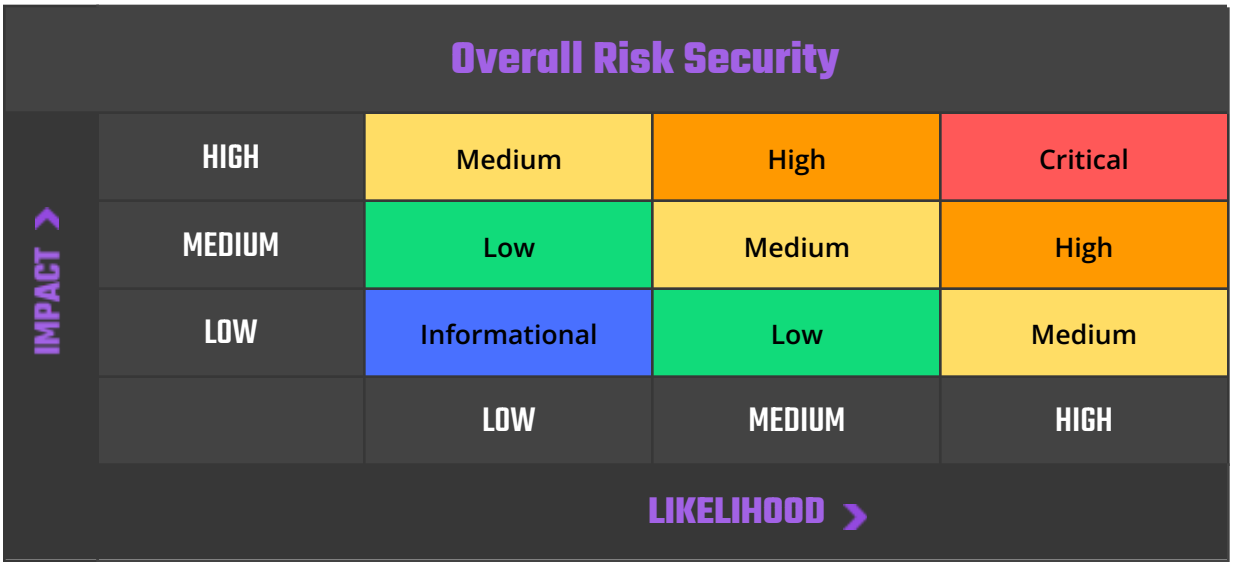
Risk Methodology

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.

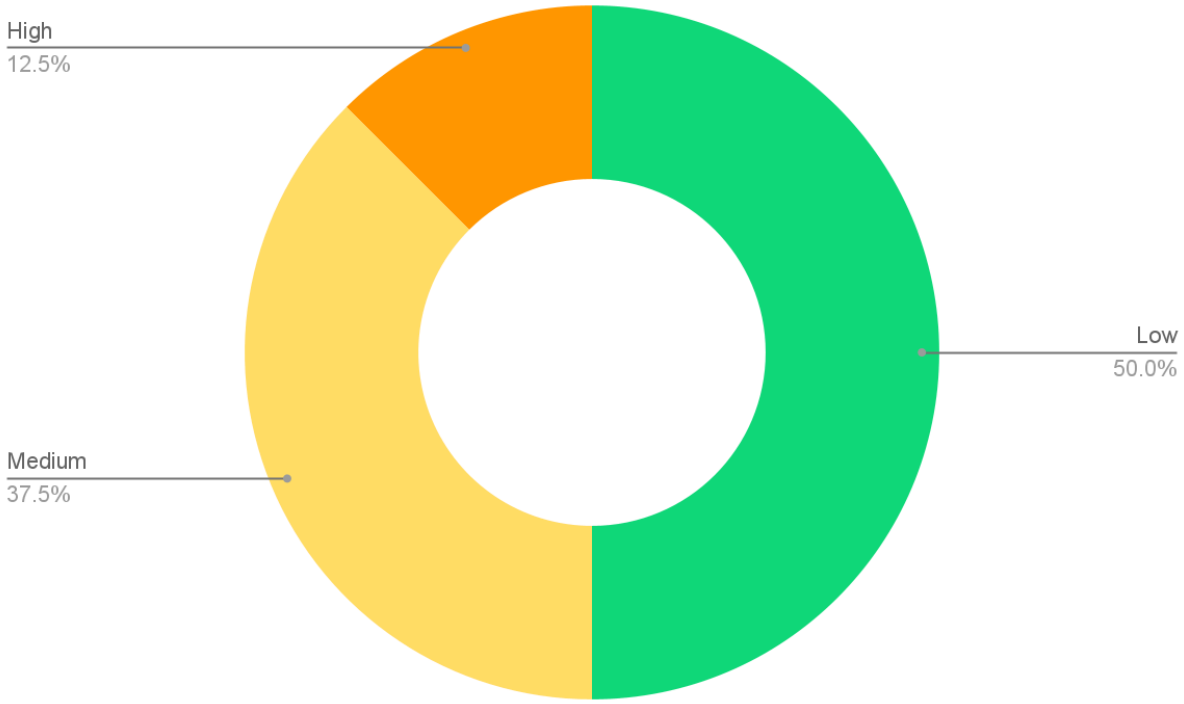
Our risk assessment model is based on two key factors: **IMPACT** and **LIKELIHOOD**. Impact refers to the potential harm that could result from an issue, such as financial loss, reputational damage, or a non-operational system. Likelihood refers to the probability that an issue will occur, taking into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



Vulnerabilities by Risk



Risk	Low	Medium	High	Critical	Informational
# of issues	4	3	1	0	0

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

The scope of the project is defined here. Our tests were performed from 03/12/2025 to 24/12/2025.

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.

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 (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).

Security Assessment Findings

Absence of Rate Limiting on Outbound Bridge Operations

ID	SAY-01
Status	Fixed
Risk	High
Business Impact	Users can initiate unlimited token transfers across chains without any rate limiting, potentially enabling rapid fund extraction in case of vulnerabilities or enabling economic manipulation attacks across chains.
Location	- `GoodBridge/ ... /MessagePassingBridge.sol; _bridgeTo(address, address, uint256, uint256, BridgeService, bytes, address)`

Description	*`MessagePassingBridge`* implements comprehensive rate limiting for inbound bridge operations through *`_enforceLimits( ... )`*, which validates transfers against *`bridgeLimits`*, *`bridgeDailyLimit`*, and *`accountsDailyLimit`*.
	
	However, *`_bridgeTo( ... )`*, which handles all outbound bridge requests initiated via *`bridgeTo( ... )`*, *`bridgeToWithLz( ... )`*, and *`bridgeToWithAxelar( ... )`*, completely bypasses this protection. The function only checks whether *`isClosed`* is true before proceeding to call *`burnFrom`* on the native token and dispatching the cross-chain message. The *`bridgeLimits`*, *`accountsDailyLimit`*, and *`bridgeDailyLimit`* state variables are never consulted during outbound operations.
	• `MessagePassingBridge.sol:377-384 (`**`abridged`**`)`
	`if (isClosed) revert BRIDGE_LIMITS('closed');`
	`...`
	`nativeToken().burnFrom(from, amount);`
	
	This asymmetry means that while receiving chains can enforce limits on incoming transfers, the sending chain cannot prevent excessive outflows. An attacker exploiting any vulnerability could rapidly drain tokens from one chain without triggering any protective rate limiting mechanisms.

Mitigation	Add a call to *`_enforceLimits( ... )`* within *`_bridgeTo( ... )`* after calculating *`normalizedAmount`* to enforce consistent bidirectional rate limiting.

Incorrect Comparison

ID	SAY-02
Status	Fixed
Risk	Medium
Business Impact	The swap operation may use an incorrect fee tier for the `GD-to-reserveToken` pool, resulting in failed swaps or suboptimal execution that yields fewer native tokens than expected during gas procurement operations.
Location	- `GoodProtocol/ ... /GenericDistributionHelper.sol; buyNativeWithGD(uint256, uint256)`

Description	*`buyNativeWithGD( ... )`* retrieves available Uniswap V3 pools for both the *`reserveToken-to-nativeToken`* pair and the *`reserveToken-to-gasToken`* pair. The function iterates through these pools to find the minimum fee tier for optimal swap execution.
	
	In the first loop iterating over *`gasPools`*, the code correctly updates *`gasFee`* by comparing the current fee against the existing value. However, in the second loop iterating over *`gdPools`*, the code incorrectly compares the local fee variable against *`gasFee`* rather than *`gdFee`*, and assigns the result to *`gdFee`*.
	• `GenericDistributionHelper.sol:301-304`
	`for (uint i = 1; i < gdPools.length; i++) {`
	`  uint24 fee = IUniswapV3Pool(gdPools[i]).fee();`
	`  gdFee = gasFee < fee ? gasFee : fee;`
	`}`
	
	This means *`gdFee`* will be set to whichever is smaller between *`gasFee`* and the current pool fee, rather than tracking the minimum fee across all GD pools. The resulting *`gdFee`* value used in the swap path encoding will likely be incorrect, causing the *`exactInput`* call to the Uniswap router to either fail due to non-existent pool or execute through a suboptimal fee tie

Mitigation	Modify line 303 to compare fee against *`gdFee`* instead of *`gasFee`*, changing the expression to correctly track the minimum fee tier across the *`gdPools`* array.

Unchecked ERC20 Transfer Return Value

ID	SAY-03
Status	Fixed
Risk	Medium
Business Impact	Token withdrawals may silently fail for non-standard ERC20 tokens that return false on failure instead of reverting, leading to loss of funds remaining locked in the contract while the transaction completes successfully.
Location	- `GoodBridge/ ... /MessagePassingBridge.sol:266; withdraw(address, uint256)`
Description	*`withdraw( ... )`* allows extraction of ERC20 tokens from the bridge contract. The function accepts a token address and amount parameter, defaulting to the full contract balance if the amount is zero. The actual transfer is executed via a direct call to the transfer method on the IERC20 interface. This call does not verify the boolean return value of the transfer operation. While modern ERC20 implementations typically revert on failure, several widely-used tokens return false instead. When interacting with such tokens, a failed transfer would not cause the transaction to revert, and the caller would receive no tokens despite the function completing without error. This can create a discrepancy between the expected behavior and actual token movement.
Mitigation	Wrap the transfer call using OpenZeppelin's *`safeTransfer( ... )`*, found in their `SafeERC20` library, or explicitly require that the return value equals true before completing execution.

Global Trading Limit Uses Symmetric Bounds unlike Directional Limits

ID	SAY-04
Status	Acknowledged
Risk	Medium
Business Impact	Protocol administrators cannot configure different thresholds for global inflows versus outflows, reducing flexibility in risk management compared to the L0 and L1 limit tiers that support asymmetric directional bounds.
Location	- `mento-core/contracts/libraries/TradingLimits.sol; verify(ITradingLimits.State, ITradingLimits.Config)`

Description	The *`TradingLimits`* library implements three limit tiers identified by flags *`L0`*, *`L1`*, and *`LG`*. For *`L0`* and *`L1`*, *`validate(ITradingLimits.Config)`* checks separate *`limit0In`*, *`limit0Out`*, *`limit1In`*, and *`limit1Out`* configuration parameters, and *`verify( ... )`* enforces these directional limits independently by comparing *`netflow0`* against *`limit0In`* for positive flows and against negative *`limit0Out`* for negative flows.
	• `TradingLimits.sol:97-104`
	`if ((config.flags & L0) > 0) {`
	`  if ((-1 * config.limit0Out) > self.netflow0) {`
	`  revert("L0Out Exceeded");`
	`  }`
	`  if (self.netflow0 > config.limit0In) {`
	`  revert("L0In Exceeded");`
	`  }`
	`}`
	
	However, the LG global limit tier uses only a single `limitGlobal` parameter. *`verify( ... )`* checks whether *`netflowGlobal`* falls outside the symmetric range of negative *`limitGlobal`* to positive *`limitGlobal`*.
	• `TradingLimits.sol:113-118`
	`if (`
	`  (config.flags & LG) > 0 &&`
	`  (-1 * config.limitGlobal > self.netflowGlobal || self.netflowGlobal > config.limitGlobal)`
	`) {`
	`  revert("LG Exceeded");`
	`}`

	validate(...) only verifies that *limitGlobal* is greater than zero when LG is enabled, without any directional parameters. • TradingLimits.sol:65 `require(self.flags & LG == 0 || self.limitGlobal > 0, "limitGlobal can't be zero if active");` This design inconsistency means that while *L0* and *L1* can be configured with different thresholds for buying versus selling pressure, the global lifetime limit cannot. In scenarios where the protocol needs to allow more cumulative inflows than outflows or vice versa over the lifetime of the limit, this symmetric constraint prevents optimal configuration.
Mitigation	Consider extending the *Config* struct to include *limitGlobalIn* and *limitGlobalOut* parameters, updating both *validate(...)* and *verify(...)* to handle asymmetric global limits consistent with the *L0* and *L1* implementation pattern.

whitelistedCount Isn't Decremented When Users Are Blacklisted

ID	SAY-05
Status	Fixed
Risk	Low
Business Impact	The *whitelistedCount* state variable becomes progressively inflated as users are blacklisted, causing any external contracts or systems relying on this counter for calculations or statistics to operate on incorrect data.
Location	- GoodProtocol/IdentityV3.sol; addBlacklisted(address)

Description	The IdentityV3 contract maintains a *whitelistedCount* state variable that tracks the total number of whitelisted addresses. *_addWhitelisted(address, uint256)* correctly increments this counter when setting an account status to 1.
	• IdentityV3.sol:313
	`whitelistedCount += 1;`
	
	_removeWhitelisted(address, uint256) also correctly decrements the counter when an account with status 1 or 2 is removed.
	• IdentityV3.sol:352
	`whitelistedCount -= 1;`
	
	However, *addBlacklisted(...)* directly sets the account status to 255 without checking the previous status value and without decrementing the counter. This creates a permanent discrepancy where the counter includes users who are no longer whitelisted.
	• IdentityV3.sol:236-241
	`function addBlacklisted(`
	`  address account`
	`) public onlyRole(IDENTITY_ADMIN_ROLE) whenNotPaused {`
	`  identities[account].status = 255;`
	`  emit BlacklistAdded(account);`
	`}`

Mitigation	Modify *addBlacklisted(...)* to check if the account status equals 1 or 2 before setting it to 255, and decrement *whitelistedCount* accordingly.

Bridge Messages from Untrusted Sources Are Not Marked as Executed

ID	SAY-06
Status	Fixed
Risk	Low
Business Impact	If an untrusted source chain or contract later becomes trusted, all previously rejected messages from that source can be replayed and executed, potentially processing stale or duplicate transactions.
Location	- `GoodBridge/ ... /MessagePassingBridge.sol; _bridgeFrom(address, address, uint256, uint256, address, uint256, BridgeService)`
Description	*`_bridgeFrom( ... )`* validates incoming bridge messages by comparing the *`sourceContract`* address against the *`trustedRemoteLookup`* mapping for the corresponding source chain. When the validation fails due to mismatched source address, empty trusted remote configuration, or length discrepancy, the function emits a *`FalseSender`* event and returns early without reverting. The function does not set *`executedRequests`* for the message *`requestId`* to true before returning, so the message remains in an unprocessed state indefinitely. • `MessagePassingBridge.sol:499-506` `if ( sourceUA.length ≠ trustedRemote.length || trustedRemote.length == 0 || keccak256(sourceUA) ≠ keccak256(trustedRemote) ) { emit FalseSender(sourceChainId, sourceContract); return; }` If protocol governance later adds the previously untrusted source to *`trustedRemoteLookup`* via *`addLzChainSupport(uint16, uint256, address)`*, any accumulated messages from that source could be replayed through the bridge. These messages would then pass trust validation and be processed, despite potentially containing outdated amounts or targeting addresses that have changed state since the original bridge request. For instance, the original `from` address could have since been blacklisted or the economic conditions may have changed.

Mitigation	Set *executedRequests* with the id parameter to true before the early return statement following the *FalseSender* event emission, preventing any future processing of the rejected message regardless of trust configuration changes.

Bridge Fee Accepts Values Exceeding 100%

ID	SAY-07
Status	Fixed
Risk	Low
Business Impact	Setting *bridgeFees.fee* to a value greater than 10,000 causes arithmetic underflow in the token distribution calculation, resulting in complete denial of service for all incoming bridge transfers.
Location	- GoodBridge/ ... /MessagePassingBridge.sol:230; setBridgeFees(BridgeFees)

Description	setBridgeFees(...) allows addresses with owner or guardian privileges to update *bridgeFees* by passing a *BridgeFees* struct. The function performs no validation on the fee field within this struct before storing it.
	• MessagePassingBridge.sol:230-233
	`function setBridgeFees(BridgeFees memory fees) external {`
	`  _onlyOwnerOrGuardian();`
	`  bridgeFees = fees;`
	`}`
	
	The fee value is subsequently used in *_takeFee(uint256)*, which calculates the fee as the amount multiplied by *bridgeFees.fee* divided by 10,000.
	• MessagePassingBridge.sol:421-423
	`function _takeFee(uint256 amount) internal view returns (uint256 fee) {`
	`  fee = (amount * bridgeFees.fee) / 10000;`
	`}`
	
	Then, in _bridgeFrom(...), the calculated fee is subtracted from tokenAmount before minting tokens to the target address.
	• MessagePassingBridge.sol:530
	`dao.mintTokens(tokenAmount - fee, target, avatar);`
	
	If *bridgeFees.fee* exceeds 10,000, the calculated fee will exceed the *tokenAmount*, causing the subtraction at line 530 to underflow. While Solidity 0.8.x will revert on underflow, this effectively creates a denial of service condition where no incoming bridge transfers can complete. Any tokens burned on the source chain would become permanently unrecoverable as the receiving side cannot process the mint operation.

Mitigation	Add a *require* statement in *setBridgeFees* to validate that the fee field within the fees parameter does not exceed 10000, ensuring the fee percentage cannot exceed one hundred percent.

External Call Execution without Reentrancy Protection

ID	SAY-08
Status	Fixed
Risk	Low
Business Impact	A malicious recipient contract could reenter *`onDistribution( ... )`* during the distribution loop, potentially receiving multiple distributions in a single transaction or manipulating contract state between iterations.
Location	- `GoodProtocol/ ... /GenericDistributionHelper.sol; onDistribution(uint256)`
Description	*`onDistribution( ... )`* iterates over *`distributionRecipients[]`* and calls *`distribute(DistributionRecipient, uint256)`* for each recipient with a non-zero bps allocation. • `GenericDistributionHelper.sol:158-166` *`distribute( ... )`* performs external calls based on the recipient *`transferType`*: for *`LayerZeroBridge`* recipients, it calls the bridge contract's *`bridgeToWithLz( ... )`*. • `GenericDistributionHelper.sol:213-233` (**abridged**)

```
uint256 totalDistributed;
for (uint256 i = 0; i < distributionRecipients.length; i++) {
   DistributionRecipient storage r = distributionRecipients[i];
   if (r.bps > 0) {
       uint256 toTransfer = (toDistribute * r.bps) / 10000;
       totalDistributed += toTransfer;
       if (toTransfer > 0) distribute(r, toTransfer);
   }
}
```

```
if (_recipient.transferType == TransferType.LayerZeroBridge) {
                                    ...
   getBridge().bridgeToWithLz{ value: lzFee }(
       _recipient.addr,
       _recipient.chainId,
       _amount,
       ""
   );
}
```

For *TransferAndCall* type recipients, it calls *transferAndCall(address, uint256, bytes)* on the native token.

- GenericDistributionHelper.sol:233-235

```
else if (_recipient.transferType == TransferType.TransferAndCall) {
   nativeToken().transferAndCall(_recipient.addr, _amount, "");
}
```

For *Transfer* type recipients it just calls *transfer(address, address, uint256, bytes)*.

- GenericDistributionHelper.sol:235-237

```
else if (_recipient.transferType == TransferType.Transfer) {
   nativeToken().transfer(_recipient.addr, _amount);
}
```

None of these code paths are protected by a reentrancy guard. *transferAndCall(...)* in particular executes the *onTokenTransfer* callback on the recipient address, giving a malicious contract full control flow before the distribution loop continues. The *totalDistributed* variable is accumulated after each *distribute(...)* call but before the state is finalized.

A reentrant call to *onDistribution(...)* would recalculate *toDistribute* based on the current *nativeToken* balance, which may have changed during the ongoing distribution. The contract inherits from *DAOUpgradeableContract* and *AccessControlEnumerableUpgradeable*, neither of which provides reentrancy protection.

The absence of OpenZeppelin's *ReentrancyGuard* or equivalent mutex mechanism leaves the distribution logic vulnerable to state manipulation.

Mitigation	Add OpenZeppelin's *ReentrancyGuard* to the contract inheritance and apply the *nonReentrant* modifier to the *onDistribution* function to prevent recursive calls during the distribution process.



We are available at security@sayfer.io

If you want to encrypt your message please use our public PGP key:

https://sayfer.io/pgp.asc

Key ID: 9DC858229FC7DD38854AE2D88D81803C0EBFCD88

Website: https://sayfer.io

Public email: info@sayfer.io

Phone: +972-559139416