

Smart Contract Audit Report for Coti

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 7
Security Evaluation **8**
Security Assessment Findings **15**
onTokenReceived(...) is Bricked 15
Unbounded nativeCotiFee Allows Full ERC20 Bridge DoS 16
Extra Operators Survive Ownership Transfer 17
No On-Chain Accounting for Outstanding Collateral 18
ERC20 Collateral Drain Through rescueERC20(...) 19
Coti Collateral Drain through rescueCoti 20
Pause Locks Users But Still Allows Admin Outflows 21
USDC.e Blacklist Risk Can Strand Bridge or User Funds 22
Public token rescue can create ghost private supply 23
Permanently Stranded Fees 24
Limits May Trap Users below Minimum Withdrawal 25
Deposit limits Apply before Actual Received Delta 26
Inconsistent Documentation and Event Metadata 27

Management Summary

Coti contacted Sayfer to perform a security audit on their smart contracts in 04/2026.

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 Coti's smart contracts.

Over the research period of 3 weeks, we discovered 13 vulnerabilities in the contract.

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

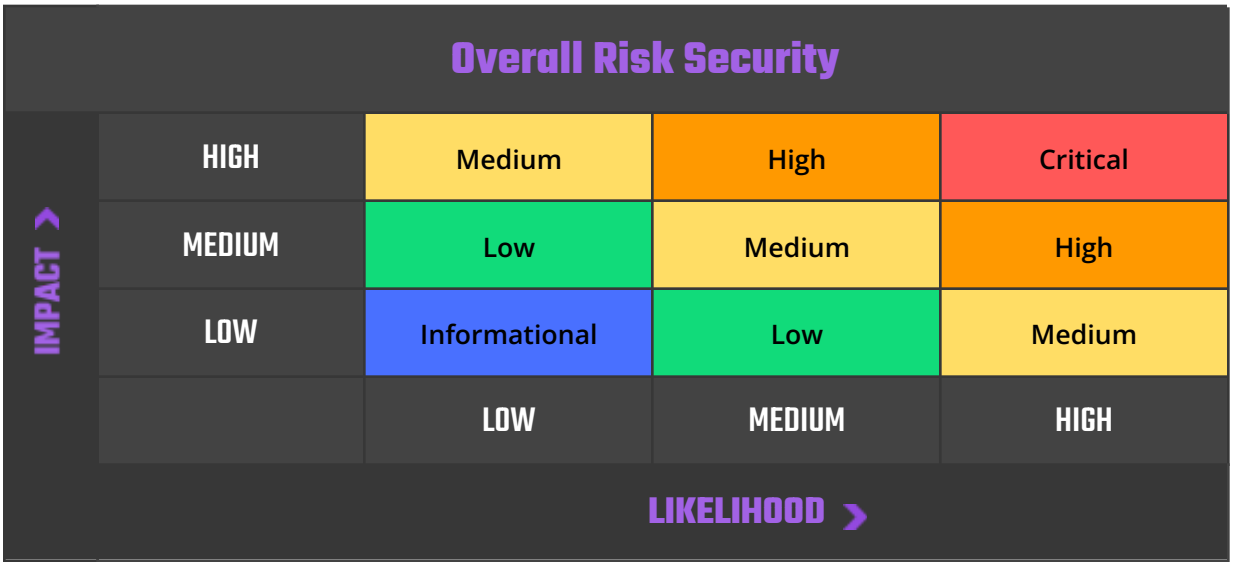
Risk Methodology

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.

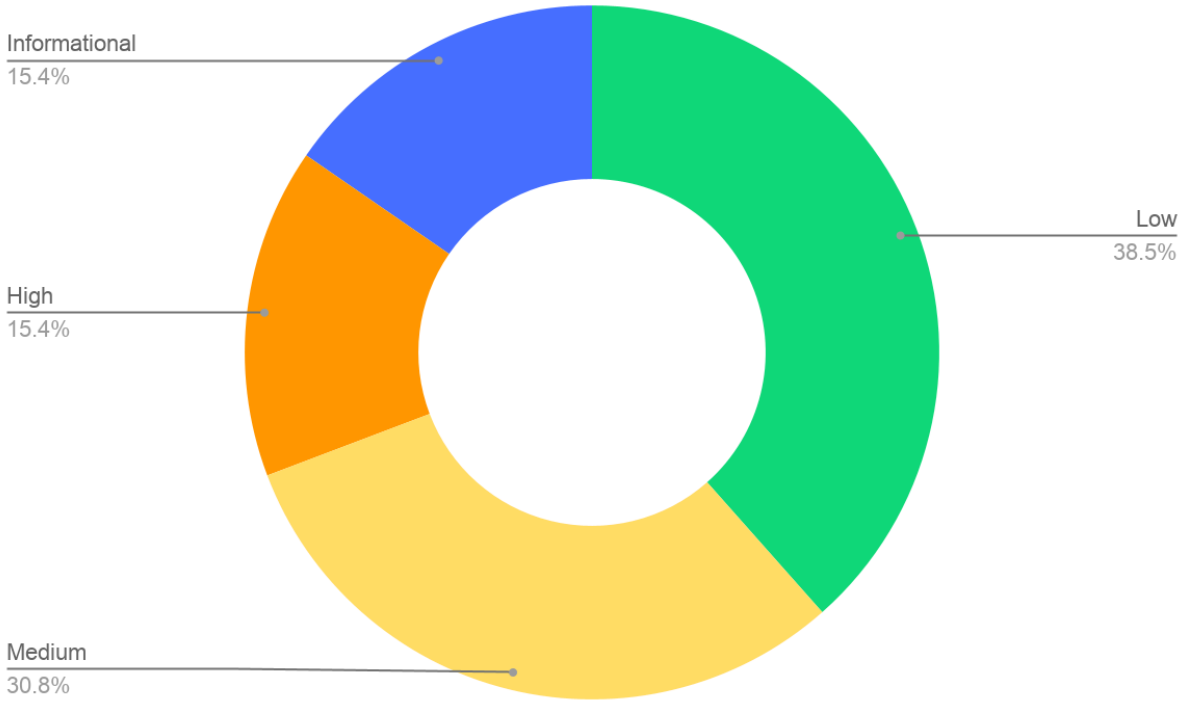
Our risk assessment model is based on two key factors: **IMPACT** and **LIKELIHOOD**. Impact refers to the potential harm that could result from an issue, such as financial loss, reputational damage, or a non-operational system. Likelihood refers to the probability that an issue will occur, taking into 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	5	4	2	0	2

Approach

Introduction

Coti 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 hash: `32e9740665ece1fa1e05172f5e64065647b020fe`

Contract	SHA-256
`PrivacyBridge.sol`	`02ad7b266aa67e06c6c9ea9baa365b0339b11200d6e5cdf71372997f737930da`
`PrivacyBridgeCotiCoti.sol`	`2973b11325ff153a56ffcf534d3008587a8b4babfbc99e5c0dbec84d252dbe5e`
`PrivacyBridgeERC20.sol`	`c4ab75129b005e767d45e7a41f4507c38b25f1e8317073feac56979e575eb8f9`
`PrivacyBridgegCoti.sol`	`6465a5869f5590d13ed5f0bc0a1b13a63ae4f19f8da8a18512111d87a39fb48b`
`PrivacyBridgeUSDCe.sol`	`d2fbaf9d911f58fe1a0d447c6e8513f807eaf59ba5bcc72dfea8b215ae2b7aac`
`PrivacyBridgeUSDT.sol`	`103c14eff2623ca4c70440ed19056689fb6e5a09ca683f220afdf4c7b25233b8`
`PrivacyBridgeWADA.sol`	`3941705e98c6adfb7892fdccd73064e23cb29cb531b63f001424e3c2b6ed0867`
`PrivacyBridgeWBTC.sol`	`f8dbaaf849925bafe61920820e1176d215875411a6fa888810daaab8b7a92eda`
`PrivacyBridgeWETH.sol`	`aa1cb033be1decb0d0c9afd2ab9a9d0b34ba88f8b681369c803c99f591804571`

Our tests were performed from 14/04/2026 to 05/05/2026.

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.

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

`onTokenReceived( ... )` is Bricked

ID	SAY-01
Status	Open
Risk	High
Business Impact	The advertised one transaction withdrawal path for native COTI always reverts. Users must fall back to the two step `approve( ... )` plus `withdraw( ... )` flow and will lose gas on the callback path.
Location	- `privacyBridge/PrivacyBridgeCotiCoti.sol; onTokenReceived(address, uint256, bytes)` - `token/PrivateERC20/PrivateERC20.sol; transferAndCall(address, uint256, bytes)`
Description	`transferAndCall( ... )` is nonReentrant and invokes the bridge callback while the token lock is still held. `onTokenReceived( ... )` then calls `privateCoti.burn(amount)`, and that function is also nonReentrant on the same token instance. That second guard acquisition always reverts because `transferAndCall( ... )` never releases the lock before entering the callback.
Mitigation	Add a bridge only burn path that does not reuse the public nonReentrant entry point, or deprecate the callback withdrawal flow entirely. The current callback path should not be documented as supported.

Unbounded `nativeCotiFee` Allows Full ERC20 Bridge DoS

ID	SAY-02
Status	Open
Risk	High
Business Impact	Any operator can set an impossible native fee and brick every ERC20 deposit and withdrawal. If a stale operator exists, the bridge can be frozen even after ownership transfer.
Location	- `PrivacyBridge.sol; setCotiCotiFee(u256)` - `PrivacyBridgeERC20.sol; _collectCotiFee()`

Description	`setCotiCotiFee(u256)` writes any `uint256` value into `nativeCotiFee` with no maximum bound. Every ERC20 deposit and withdrawal later checks `msg.value < nativeCotiFee` inside `_collectCotiFee()`, so setting the fee near `type(uint256).max` makes the condition impossible for realistic transactions to satisfy.
	● `PrivacyBridge.sol:253-256`
	`function setCotiCotiFee(uint256 _fee) external onlyOperator {`
	`  nativeCotiFee = _fee;`
	`  emit CotiCotiFeeUpdated(_fee, msg.sender);`
	`}`

Mitigation	Add a reasonable hard cap for `nativeCotiFee` and revert above that limit. Pair the cap with stale operator cleanup during ownership transfer.

Extra Operators Survive Ownership Transfer

ID	SAY-03
Status	Open
Risk	Medium
Business Impact	A previously added operator retains fee control, deposit toggles, and fee withdrawal access after ownership moves to a new party. This leaves a hidden privileged actor in place and expands the attack surface.
Location	- `PrivacyBridge.sol; transferOwnership(address)`

Description	`transferOwnership( ... )` revokes `DEFAULT_ADMIN_ROLE` and `OPERATOR_ROLE` only from the current owner. It does not enumerate or revoke any other account that previously received `OPERATOR_ROLE` through `addOperator(address)`. Because the bridge uses non-enumerable `AccessControl`, the new owner cannot discover all active operators from on chain state alone and must reconstruct events off chain.
	• `PrivacyBridge.sol:140-148`
	`function transferOwnership(address newOwner) public override onlyOwner {`
	`  if (newOwner == address(0)) revert InvalidAddress();`
	`  address oldOwner = owner();`
	`  super.transferOwnership(newOwner);`
	`  _grantRole(DEFAULT_ADMIN_ROLE, newOwner);`
	`  _grantRole(OPERATOR_ROLE, newOwner);`
	`  _revokeRole(DEFAULT_ADMIN_ROLE, oldOwner);`
	`  _revokeRole(OPERATOR_ROLE, oldOwner);`
	`}`

Mitigation	Move to `AccessControlEnumerable` or maintain your own operator registry so ownership transfer can revoke or review all operators. Ownership transfer should reset operator state to a known clean set.

No On-Chain Accounting for Outstanding Collateral

ID	SAY-04
Status	Open
Risk	Medium
Business Impact	The bridge cannot enforce solvency on chain because it does not know how much public collateral is still owed to private token holders. Rescue and operational checks therefore rely on trust instead of enforceable invariants.
Location	- `privacyBridge/PrivacyBridge.sol` - `token/PrivateERC20/PrivateERC20.sol; totalSupply()`

Description	The private token intentionally returns 0 from totalSupply, which means the bridge cannot use token supply as a measure of liability.
	• `PrivateERC20.sol:202-204`
	`function totalSupply() public view virtual override returns (uint256) {`
	`  return 0;`
	`}`
	
	The bridge state tracks fee counters and limits, but it does not track any variable that increases on deposit net minting and decreases on withdrawal net burning. Without that state, the bridge cannot prove or enforce backing balance >= outstanding user liability + fees.

Mitigation	Track outstanding collateral directly in the bridge using the net amount minted and net amount released. Use that value in all rescue and liquidity calculations.

ERC20 Collateral Drain Through `rescueERC20( ... )`

ID	SAY-05
Status	Open
Risk	Medium
Business Impact	The owner can transfer the backing ERC20 asset out of the bridge while private liabilities remain outstanding. A malicious or compromised admin can leave all ERC20 private balances unbacked and permanently break withdrawals.
Location	- `PrivacyBridgeERC20.sol; rescueERC20(address, address, uint256)`

Description	`rescueERC20( ... )` blocks only `address(privateToken)` and does not block `address(token)`. The function therefore allows the owner to move the actual public collateral asset used to satisfy withdrawals.
	• `PrivacyBridgeERC20.sol:246-247`
	`if ( _token == address(privateToken))`
	`revert CannotRescueBridgeToken();`
	
	The bridge also does not track an `outstandingCollateral` value, so there is no on chain reserve check linking the bridge balance to the net amount minted in `_deposit(uint256, bool, itUint256)`.

Mitigation	Block rescue of `address(token)` and add an explicit reserve invariant based on tracked outstanding collateral. Treat rescue and fee withdrawal as surplus only operations and consider disabling them while the bridge is paused.

Coti Collateral Drain through rescueCoti

ID	SAY-06
Status	Open
Risk	Medium
Business Impact	The owner can withdraw native balance that still backs outstanding private COTI. A malicious or compromised admin can fully unback p.COTI and leave all later withdrawals insolvent.
Location	- `PrivacyBridgeCotiCoti.sol; rescueCoti(address, uint256)`
Description	`rescueCoti( ... )` reserves only `accumulatedFees` and does not reserve user liabilities. The current check allows the owner to withdraw any native balance above the fee counter even though the remaining balance may still be needed for future withdrawals or `onTokenReceived(address, uint256, bytes)` redemptions. This is the same structural issue as the ERC20 rescue path, except here the allowed amount is derived from `address(this).balance - accumulatedFees`.
Mitigation	Introduce tracked outstanding collateral and bound rescue to true surplus over user liabilities plus fees. If rescue is retained, gate it with stricter operational controls and strongly consider blocking it during pause.

Pause Locks Users But Still Allows Admin Outflows

ID	SAY-07
Status	Open
Risk	Low
Business Impact	Users lose all normal exit paths during pause while owner and operators keep access to rescue and fee withdrawal.

Location	- `PrivacyBridge.sol`
	  - `withdrawCotiFees(address, uint256)`
	- `PrivacyBridgeERC20.sol`
	  - `deposit(uint256)`
	  - `withdraw(uint256)`
	  - `rescueERC20(address, address, uint256)`
	  - `withdrawFees(address, uint256)`
	- `PrivacyBridgeCotiCoti.sol`
	  - `withdraw(uint256)`
	  - `onTokenReceived(address, uint256, bytes)`
	  - `receive()`
	  - `rescueERC20(address, uint256)`
	  - `withdrawFees(address, uint256)`

Description	Many user-facing functions use the modifier `whenNotPaused`, including ERC20 `deposit(uint256)` and `withdraw(uint256)`, native `withdraw(uint256)` `onTokenReceived( ... )` and `receive()`. Meanwhile, admin outflow functions such as `rescueERC20( ... )`, `rescueCoti( ... )`, `withdrawFees( ... )` and `withdrawCotiFees( ... )` do not employ the modifier and therefore will continue to function during a pause. Additionally, pausing has no expiry date and there is no emergency user exit.
Mitigation	Decide whether pause is an emergency stop or an admin priority state and encode that choice consistently. If users are blocked, rescue and fee withdrawal should normally be blocked too, unless a carefully designed emergency exit exists.

USDC.e Blacklist Risk Can Strand Bridge or User Funds

ID	SAY-08
Status	Open
Risk	Low
Business Impact	If the bridged USDC.e token blacklists the bridge, deposits, withdrawals, and rescue operations can all stop. If a user address is blacklisted, that user may be unable to receive redeemed USDC.e through the current API.

Location	- `PrivacyBridgeERC20.sol`
	  - `_deposit(uint256, bool, itUint256)`
	  - `_withrdaw(uint256, bool, itUint256)`
	  - `rescueERC20(address, address, uint256)`

Description	The bridge pulls the public token in `_deposit( ... )`, sends it out in `_withrdaw( ... )`, and sends it out again in `rescueERC20( ... )`. A blacklistable fiat backed token can reject those transfers when either the bridge or recipient is blocked. The bridge has no alternative delivery path such as `withdrawTo( ... )`, and the native bridge style rescue logic does not burn private balances when public tokens are diverted. For a blacklistable deployment asset, this is a real protocol dependency rather than a coding mistake.
Mitigation	Document the blacklist dependency explicitly and plan an operational response for blocked bridge or user addresses. If the intended asset is USDC.e, add a recipient override flow for user initiated withdrawals.

Public token rescue can create ghost private supply

ID	SAY-09
Status	Open
Risk	Low
Business Impact	If public collateral is manually rescued for a specific user case, the corresponding private balance can remain outstanding. Later users may then burn valid private balances against a bridge that no longer holds enough public collateral.
Location	- `PrivacyBridgeERC20.sol; rescueERC20(address, address, uint256)`
Description	`rescueERC20( ... )` moves public tokens out of the bridge and does not interact with the paired private token at all. When the rescued asset is the actual backing token, this removes collateral without reducing any private balance. Because the bridge also lacks outstanding collateral accounting, the solvency gap is invisible on-chain. This becomes especially relevant in blacklist remediation scenarios where the owner may try to manually redirect collateral.
Mitigation	Do not use the generic rescue path for backing assets. If exceptional user specific recovery is needed, build a dedicated flow that burns or otherwise settles the matching private balance.

Permanently Stranded Fees

ID	SAY-10
Status	Open
Risk	Low
Business Impact	The fee counter can remain positive while the actual bridge token balance is already gone. Fee recipients then see nonzero `accumulatedFees` that can never be withdrawn.
Location	- `PrivacyBridgeERC20.sol; withdrawFees(address, uint256)`
Description	`withdrawFees( ... )` only checks that the amount to be paid is smaller or equal to the accumulated fees before attempting the ERC20 transfer. • `PrivacyBridgeERC20.sol:225` `if (amount > accumulatedFees) revert InsufficientAccumulatedFees();` `rescueERC20(address, address, uint256)` can later drain the same public token balance without reducing `accumulatedFees`. Once the balance is gone, `withdrawFees( ... )` will keep passing the counter check and failing at the transfer, which reverts the counter decrement.
Mitigation	Add a direct balance check before transferring fees and make sure collateral draining paths cannot leave the fee counter detached from actual balances. The cleaner fix is still to ban rescue of backing assets.

Limits May Trap Users below Minimum Withdrawal

ID	SAY-11
Status	Open
Risk	Low
Business Impact	An accepted deposit can mint less private value than the configured minimum withdrawal amount. Users then need to top up before they can redeem.
Location	- `PrivacyBridge.sol; setLimits(uint256, uint256, uint256, uint256)`

Description	`setLimits( ... )` only checks that each minimum is less than or equal to its paired maximum. It does not check that a minimum deposit remains withdrawable after deposit fee deduction.
	• `PrivacyBridge.sol:165-166`
	`if (_minDeposit > _maxDeposit) revert InvalidLimitConfiguration();`
	`if (_minWithdraw > _maxWithdraw) revert InvalidLimitConfiguration();`
	
	If `minDepositAmount` is set to a value that mints less than `minWithdrawAmount`, a user can make a valid deposit and still end up with an unredeemable balance until they deposit more.

Mitigation	Validate cross parameter coherence when limits or fees change. At minimum, reject configurations where the smallest valid deposit mints less than the smallest valid withdrawal.

Deposit limits Apply before Actual Received Delta

ID	SAY-12
Status	Open
Risk	Informational
Business Impact	For fee on transfer tokens, the bridge would validate the larger requested amount rather than the smaller actual received amount. The code comments say those token types are out of scope, so this is a specification note rather than an active bug for intended assets.
Location	- `PrivacyBridgeERC20.sol; _deposit(uint256, bool, itUint256)`
Description	`_deposit( ... )` calls `_checkDepositLimits(uint256)` before it measures the balance delta around `safeTransferFrom(address, address, uint256)`. The actual minting amount later derives from received, not from the requested amount. For standard non-rebasing and non-fee on-transfer tokens this distinction does not matter. For other token types it would, which is why the code comments explicitly forbid them.
Mitigation	Keep the current implementation only if the supported asset set stays limited to standard ERC20 tokens. If broader token support is ever introduced, apply limits to the measured received value instead.

Inconsistent Documentation and Event Metadata

ID	SAY-13
Status	Open
Risk	Informational
Business Impact	

Location	- `PrivacyBridge.sol`
	- `PrivacyBridgeERC20.sol`
	  - `withdrawFees(address, uint256)`
	- `PrivacyBridgeCotiCoti.sol`
	  - `withdrawFees(address, uint256)`

Description	`withdrawFees(address, uint256)` in both concrete bridges is restricted by the `onlyOperator` modifier, yet the NatSpec text says only the owner can call it.
	• `PrivacyBridgeCotiCoti.sol:225-230`
	`/**`
	`* @notice Withdraw accumulated fees (Coti implementation)`
	`* @param to Address to send fees to`
	`* @param amount Amount of fees to withdraw`
	`* @dev Only the owner can call this function`
	`*/`
	
	The fee scale documentation also mixes micro percent language with standard basis point language, which can cause operator misconfiguration.
	
	In addition, the events `LimitsUpdated` and `FeeUpdated` do not include the actor field even though some other administrative events do.

Mitigation	• Standardize fee unit documentation and align the NatSpec comment for each function with the actual modifier that enforces access to it. • Add caller attribution to administrative events that currently omit it.



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