

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 6
Security Evaluation **7**
Security Assessment Findings **14**
Inverted Allowance Deduction Logic 14
Single Transaction Drain 16
Unlimited Minting 18
Unchecked MPC Transfer Return 20
Incompatibility with ERC20 21
Missing Event Emission 23

Management Summary

Coti contacted Sayfer to perform a security audit on their smart contract in 03/2026.

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 contract.

Over the research period of 3 weeks, we discovered 6 vulnerabilities in the contract. Two of them are marked as critical, as deploying the contract as-is may lead to direct loss of funds and functionality.

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 Coti team.

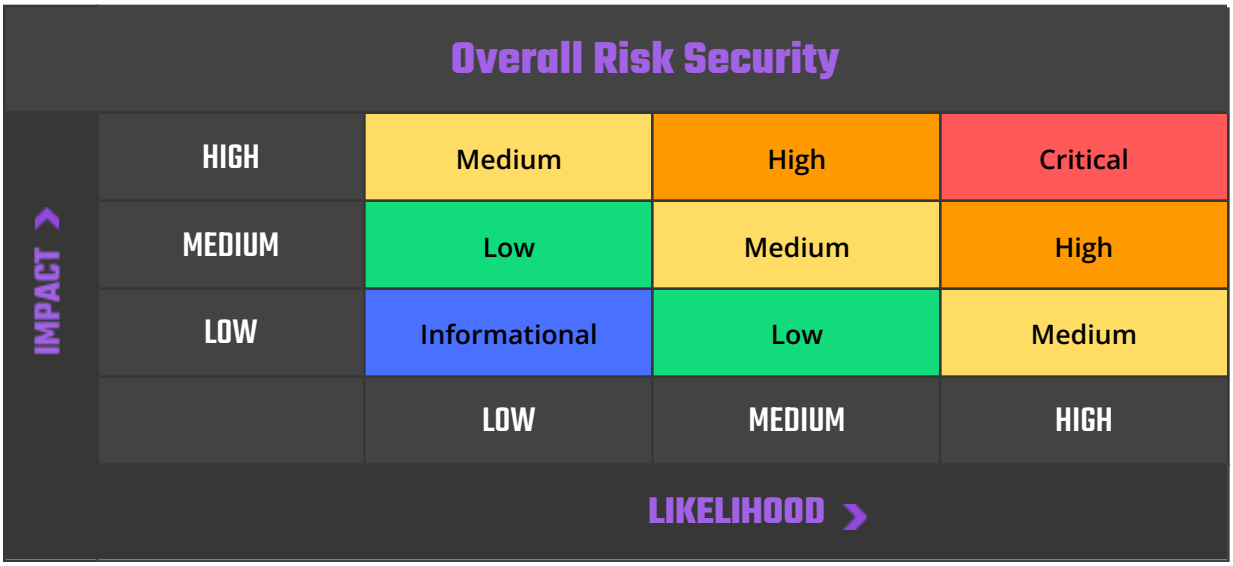
Risk Methodology

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.

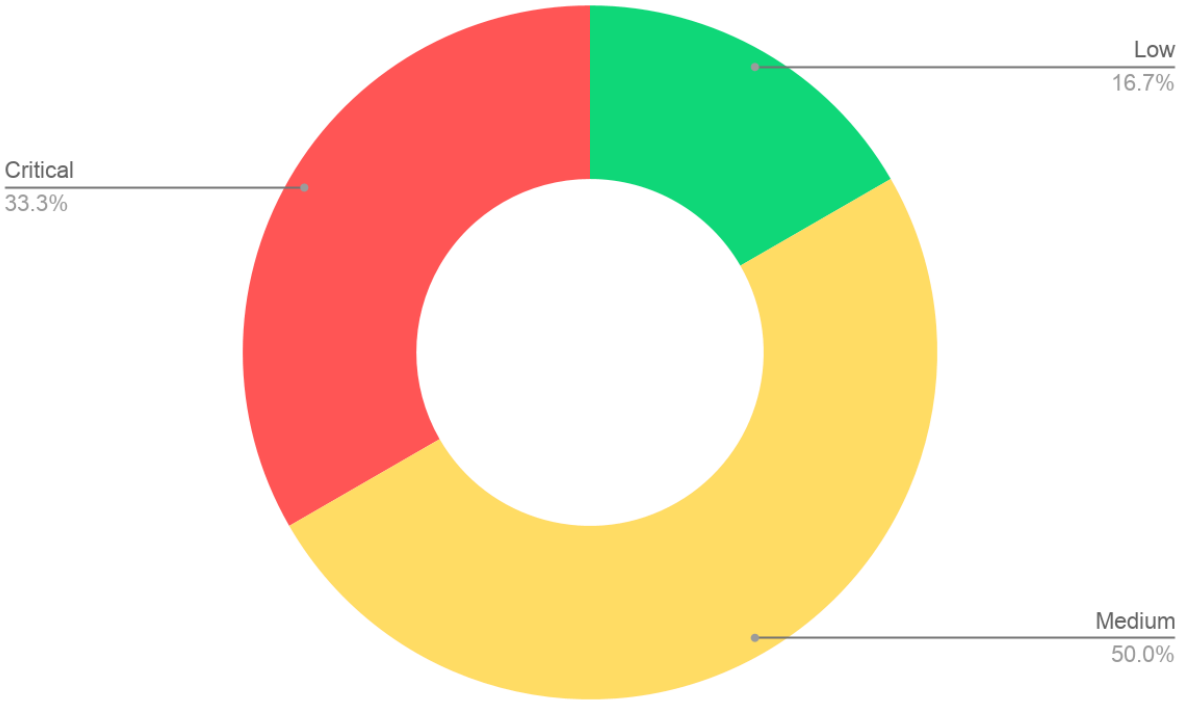
Our risk assessment model is based on two key factors: **IMPACT** and **LIKELIHOOD**. Impact refers to the potential harm that could result from an issue, such as financial loss, reputational damage, or a non-operational system. Likelihood refers to the probability that an issue will occur, taking into 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	1	3	0	2	0

Approach

Introduction

Coti contacted Sayfer to perform a security audit on their smart contract.

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.

Contract	SHA-256
`PrivateERC20.sol`	`12b67271ebd94d3797c0940296dc67e3c327d0e99263218b4e42438d37414a58`

Branch: development
Audit commit: b79cae56e7baa46a801f505fc0e7708d1b3502e6
Fixes commit: 14ba862202d0d0b77f27e90430fabd3d2bc3a086

Our tests were performed from 17/03/2026 to 31/03/2026.

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

Inverted Allowance Deduction Logic

ID	SAY-01
Status	Fixed
Risk	Critical
Business Impact	Any address approved for any amount (even a single wei!) can drain the owner's **entire** token balance through repeated `transferFrom( ... )` calls.
Location	- `PrivateERC20.sol; _spendAllowance(address, address, gtUint256)`

Description	`_spendAllowance( ... )` contains an inverted `MpcCore.mux( ... )` call that causes the allowance to never be decremented on successful `transferFrom( ... )` operations. Any approved spender can repeatedly transfer the full approved amount until the owner's entire balance is drained.
	
	The bug arises from two interacting issues. First, `_spendAllowance( ... )` is called after `_transfer(address, address, gtUint256)` has already executed (L648), so it reads the owner's post-transfer balance rather than the pre-transfer balance.
	• `PrivateERC20.sol:645-650`
	`gtBool success = _transfer(from, to, gtValue);`
	`require(MpcCore.decrypt(success), "ERC20: transfer failed");`
	`_spendAllowance(from, spender, gtValue);`
	`return true;`
	
	Second, the arguments to `MpcCore.mux( ... )` are swapped. The `MpcCore.mux(condition, a, b)` returns *a* when condition is true and *b* when condition is false. The condition represents "should we skip the deduction". For a normal valid transfer, all three flags are false, so the condition evaluates to false, and mux returns *b* - the unchanged `currentAllowance`. The decremented value in position *a* is only returned when the condition is true (the skip cases), which is the exact opposite of the intended behavior.
	• `PrivateERC20.sol:932-939`
	`gtUint256 newAllowance = MpcCore.mux(`
	`MpcCore.or(`
	`maxAllowance,`
	`MpcCore.or(insufficientBalance, inSufficientAllowance)`

```
    ),
    MpcCore.sub(currentAllowance, value),
    currentAllowance
);
```

Exploitation Scenario:
The owner, which has 1000e18 tokens, approves a spender for 100e18. Spender calls transferFrom(owner, recipient, 100e18):

- `_transfer( ... )` moves 100e18 from owner to recipient. Owner balance: 900e18.
- `_spendAllowance( ... )` reads `currentBalance = 900e18` (post-transfer), `currentAllowance = 100e18`.
- `insufficientBalance = lt(900e18, 100e18) = FALSE; insufficientAllowance = lt(100e18, 100e18) = FALSE; maxAllowance = FALSE`
- `mux(FALSE, sub(100e18, 100e18), 100e18)` returns 100e18 - allowance unchanged.
- Spender calls `transferFrom( ... )` again with the same 100e18 allowance.
- Repeat until the owner's balance is 0.

Mitigation	• Swap the arguments to `MpcCore.mux( ... )` so that the decremented allowance is returned in the normal case (`condition = false`) and the unchanged allowance is returned in the skip case (`condition = true`). • Additionally, consider calling `_spendAllowance( ... )` before `_transfer( ... )` to avoid the stale balance read, which would make the allowance check use the owner's pre-transfer balance for the `insufficientBalance` condition.

Single Transaction Drain

ID	SAY-02
Status	Fixed
Risk	Critical
Business Impact	Complete single-transaction drain of any account that has approved the malicious receiver contract. The attack is amplified beyond SAY-01 because the entire drain happens atomically – no multi-block coordination needed, making it exploitable in a flash-loan-style pattern
Location	- `PrivateERC20.sol; transferAndCall(address, uint256, bytes)`
Description	This finding combines two vulnerabilities into a single-transaction drain attack: the allowance deduction inversion described in SAY-01 and a missing reentrancy guard on `transferFrom( ... )`. The `transferAndCall( ... )` function carries the nonReentrant modifier from OpenZeppelin's `ReentrancyGuard`. However, `transferFrom( ... )` does NOT have this modifier. During the `onTokenReceived(address, uint256, bytes)` callback at L264, a malicious receiver contract gains execution control and can freely call `transferFrom( ... )` – the reentrancy lock on `transferAndCall( ... )` only prevents re-entering `transferAndCall( ... )` itself, not other functions. The NatSpec at L245 incorrectly claims "the callback cannot re-enter the token (nonReentrant)." This is factually wrong - the callback can call any function that lacks the `nonReentrant` modifier, which includes `transferFrom( ... )`, `transfer( ... )`, `approve( ... )`, `burn( ... )`, and others. The complete attack sequence in a single transaction: • Attacker deploys a malicious contract implementing `ITokenReceiver` • Victim approves the malicious contract for amount X (standard ERC20 workflow) • Any user calls `transferAndCall(maliciousContract, smallAmount, "")`, triggering the callback. • Inside `onTokenReceived( ... )`, the malicious contract calls `transferFrom(victim, attacker, X)`. This succeeds because `transferFrom( ... )` has no reentrancy guard • Due to SAY-01, the allowance is NOT decremented after this call. • The malicious contract repeats step 4 as many times as needed. • Result: victim loses N * X tokens in a single transaction, limited only by the victim's balance.

Mitigation	Apply the `nonReentrant` modifier to all public token-transfer and allowance-spending functions, not just `transferAndCall( ... )`. At minimum, add `nonReentrant` to all three `transferFrom( ... )` overloads.

Unlimited Minting

ID	SAY-03
Status	Fixed
Risk	Medium
Business Impact	A compromised or malicious admin can cause unlimited token supply inflation in a single block.
Location	- `PrivateERC20.sol; mint(address, uint256)`
Description	The `DEFAULT_ADMIN_ROLE` assigned at construction can instantly grant `MINTER_ROLE` to any address via OpenZeppelin's `grantRole( ... )`. The new `MINTER_ROLE` holder can then mint unlimited tokens with no supply cap, no rate limit, no cooldown, and no timelock on either the role grant or the minting operations. • `PrivateERC20.sol:178-189`

```
function mint(
    address to,
    uint256 amount
) public virtual override onlyRole(MINTER_ROLE) returns (bool) {
    if (to == address(0)) revert ERC20InvalidReceiver(address(0));
    if (!publicAmountsEnabled) revert PublicAmountsDisabled();
    gtUint256 gtAmount = MpcCore.setPublic256(amount);
    gtBool success = _mint(to, gtAmount);
    require(MpcCore.decrypt(success), "ERC20: mint failed");

    return true;
}
```

Furthermore, `totalSupply()` always returns 0 by design, as documented in its spec, which means there is no on-chain mechanism for smart contracts or governance systems to verify the actual token supply. While Transfer events from `address(0)` are emitted and detectable off-chain, automated on-chain supply monitoring is impossible.

- `PrivateERC20.sol:169-176`

```
/**
    * @dev See {IPrivateERC20-totalSupply}.
    *      For privacy, always returns 0; actual total supply is stored
encrypted in _totalSupply.
    *      Integrators: use for display only; do not rely on this for
supply accounting.
```

```
   */
function totalSupply() public view virtual override returns (uint256) {
   return 0;
}
```

Mitigation	Implement defense-in-depth mechanisms around the minting capability. Add an on-chain supply cap that cannot be exceeded by minting. Introduce a timelock on `grantRole( ... )` for sensitive roles like `MINTER_ROLE`.

Unchecked MPC Transfer Return

ID	SAY-04
Status	Fixed
Risk	Medium
Business Impact	Off-chain indexers may record fake transfers, while block explorers may display misleading transfer histories for affected addresses. DeFi protocol integrations that rely on Transfer events for accounting could also develop incorrect state. Additionally, phantom events leak information about failed transaction attempts, causing a breach of privacy.
Location	- `PrivateERC20.sol; _update(address, address, gtUint256)`

Description	`_update( ... )` emits a Transfer event unconditionally, regardless of whether the underlying operation succeeded or failed. This is problematic for transfer variants that do not revert upon failure.
	• `PrivateERC20.sol:762` — Note how the emission is not conditioned.
	`emit Transfer(from, to, senderCt, receiverCt);`
	
	When `MpcCore.transfer( ... )` returns false (insufficient balance), the balances remain unchanged but a `Transfer` event is still emitted. `transferGT( ... )` and `transfer(address, itUint256)` return the `gtBool` result without reverting, so the phantom event persists permanently on-chain.
	
	In contrast, `transfer(address, uint256)` correctly calls `require(MpcCore.decrypt(success))` which reverts the transaction and rolls back the event on failure.
	• `PrivateERC20.sol:423`
	`require(MpcCore.decrypt(success), "ERC20: transfer failed");`
	
	This creates a permissionless griefing vector as anyone can call `transferGT( ... )` with an amount exceeding their balance, generating phantom Transfer events targeting any address. For a privacy-focused token, event integrity is part of the core value proposition.

Mitigation	Move the emit `Transfer` inside a conditional that checks the result before emitting.

Incompatibility with ERC20

ID	SAY-05
Status	Fixed
Risk	Medium
Business Impact	Incompatibility with standard ERC20 may cause developer and user confusion and potentially hidden issues.
Location	- `PrivateERC20.sol:215, 239, 284, 395, 515, 573`
Description	Case #1: The two branches of the encrypted allowance getter both resolve to `_allowances[spender][owner]` instead of the correct `_allowances[owner][spender]` that `_approve(address, address, gtUint256)` writes. • `PrivateERC20.sol:451-461 (with added comments)` Both branches read from the uninitialized slot. In production with real MPC ciphertexts, both branches will return empty ciphertext regardless of what was approved. Case #2: The re-encryption helper compounds the allowance() key inversion and also writes to the wrong ciphertext field. • `PrivateERC20.sol:478-492 (with added comments)`

```
function allowance(
    address account,
    bool isSpender
) public virtual override returns (gtUint256) {
    if (account == address(0)) revert ERC20InvalidReceiver(address(0));
    if (isSpender) {
        // Caller = spender, account = owner
        // Reads: _allowances[_msgSender()][account] = _allowances[spender][owner]
        // Should: _allowances[account][_msgSender()] = _allowances[owner][spender]
        return _safeOnboard(_allowances[_msgSender()][account].ciphertext);
    } else {
        // Caller = owner, account = spender
        // Reads: _allowances[account][_msgSender()] = _allowances[spender][owner]
        // Should: _allowances[_msgSender()][account] = _allowances[owner][spender]
        return _safeOnboard(_allowances[account][_msgSender()].ciphertext);
    }
}
```

```
if (isSpender) {
    // Reads _allowances[_msgSender()][account] = _allowances[spender][owner]
```

```
    Allowance storage allowance_ = _allowances[_msgSender()][account];

    // Writes ownerCiphertext - caller is spender, should write spenderCiphertext
    allowance_.ownerCiphertext = MpcCore.offBoardToUser(
        _safeOnboard(allowance_.ciphertext),
        encryptionAddress
    );
} else {
    // Reads _allowances[account][_msgSender()] = _allowances[spender][owner]
    Allowance storage allowance_ = _allowances[account][_msgSender()];

    // Writes spenderCiphertext - caller is owner, should write ownerCiphertext
    allowance_.spenderCiphertext = MpcCore.offBoardToUser(
        _safeOnboard(allowance_.ciphertext),
        encryptionAddress
    );
}
```

Case #3:
The plaintext `approve(address, uint256)` does not use the `increaseAllowance` and `decreaseAllowance` pattern, leaving the standard ERC20 approve race condition in place. A spender who observes a pending `approve(spender, newValue)` transaction can front-run it to spend the old allowance before it is replaced, then spend the new allowance after, extracting `oldValue + newValue` instead of `newValue`.

Mitigation	1. Swap the argument order in both branches. When `isSpender=true`, the caller is the spender and account is the owner, so read `_allowances[account][_msgSender()]`. When `isSpender=false`, the caller is the owner and account is the spender, so read `_allowances[_msgSender()][account]`. 2. Apply the same key fix. Additionally, when `isSpender=true` write to `allowance_.spenderCiphertext`, when `isSpender=false` write to `allowance_.ownerCiphertext`. 3. Use `increaseAllowance` and `decreaseAllowance` helpers, or require the current allowance to be zero before setting a non-zero value, following the established mitigation for EIP-20.

Missing Event Emission

ID	SAY-06
Status	Fixed
Risk	Low
Business Impact	Off-chain monitoring cannot detect allowance re-encryption events. Indexers tracking allowance history will have incomplete records, potentially causing UIs to display stale ciphertext data.
Location	- `PrivateERC20.sol; reencryptAllowance(address, bool)`
Description	`reencryptAllowance(address, bool)` re-encrypts an existing allowance entry under the caller's current encryption key, modifying either `ownerCiphertext` or `spenderCiphertext` in `_allowances[address][address]`. Despite writing to persistent storage, the function emits no event. Off-chain systems, indexers, UIs, and wallets that track allowance state have no way to observe that re-encryption occurred. By contrast, the functionally similar `setAccountEncryptionAddress(address)` correctly emits an event. • `PrivateERC20.sol:362` `emit AccountEncryptionAddressSet(account, offBoardAddress);` The asymmetry is inconsistent and creates a blind spot in off-chain allowance tracking. The function is also the designated remediation step after key rotation, making silent execution especially problematic, as users who re-encrypt their allowances to recover from a key rotation leave no observable on-chain trace.
Mitigation	Emit an `AllowanceReencrypted(address indexed owner, address indexed spender, bool isSpender)` event at the end of `reencryptAllowance`, or call the existing `_approve(address, address, gtUint256)` to re-emit the updated ciphertext values.



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