SAYFER

Smart Contract Audit Report for Coti

Testers

1. Or Duan
2. Avigdor Sason Cohen

Table of Contents

Management Summary

Coti contacted Sayfer to perform a security audit on their smart contract in 10/2025.

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 Coti's smart contract.

Over the research period of 2 weeks, we discovered 10 vulnerabilities in the contract. None of them is marked as critical.

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

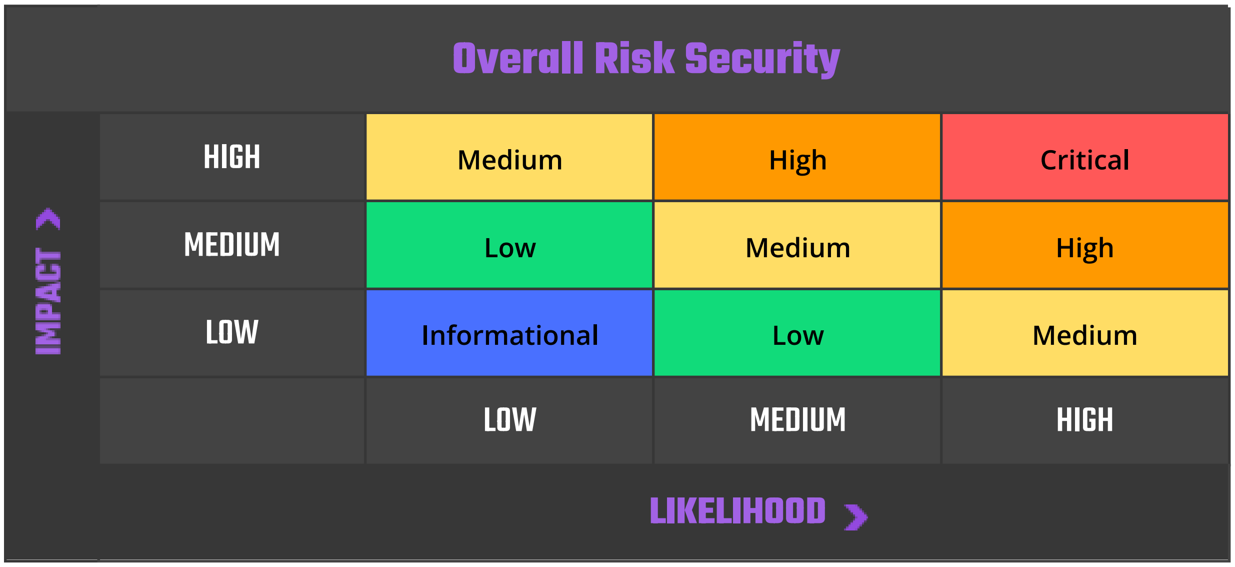
Risk Methodology

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.

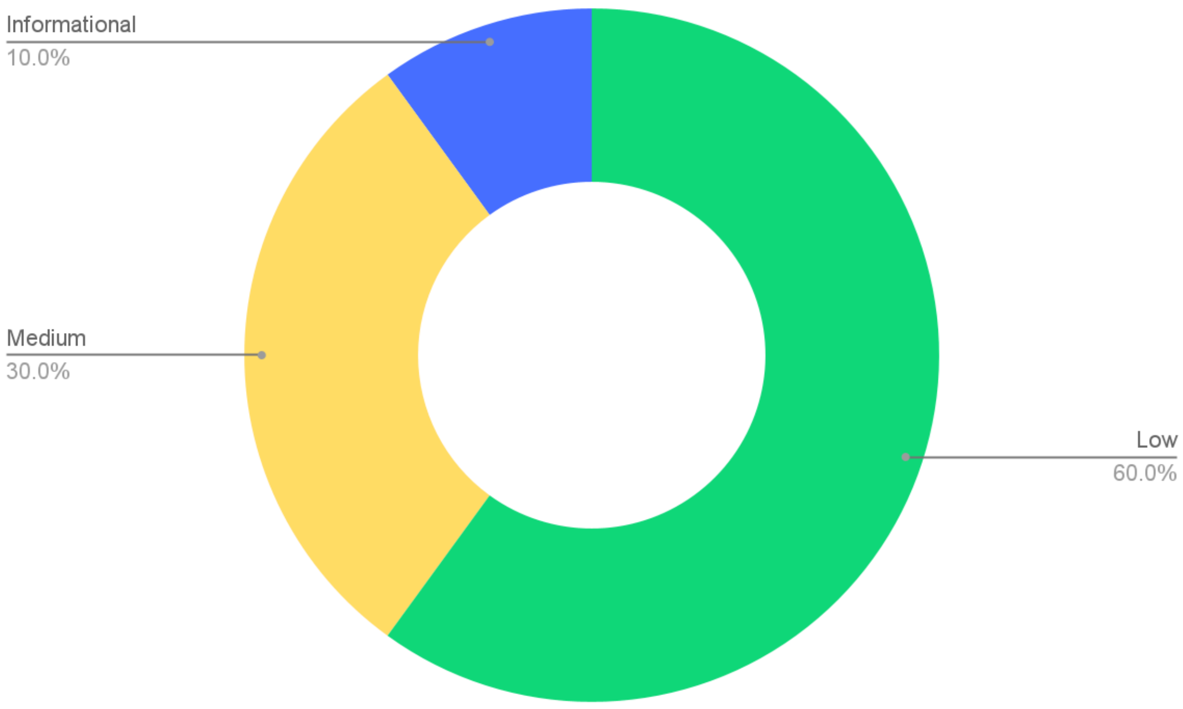
Risk is defined as follows:



Overall Risk Security

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

Vulnerabilities by Risk



Risk	Low	Medium	High	Critical	Informational
# of issues	6	3	0	0	1

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.

Commit hash: 22879a9c9a3fc69b4e17b4e2de79da3f7dc47602

Contract	SHA-256
`FixedRatioCoinDisperser.sol`	`f027f9a77c408b153a5b3a48f9a89cca1fe993381bf219f4b2c105840fb7d0c3`

Our tests were performed from 30/10/2025 to 13/11/2025.

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.

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

Owner can drain users' pending pull redemptions after the window via `rescueEth`

ID	SAY-01
Status	Open
Risk	Medium
Business Impact	Loss of funds for users who chose the pull path but have not yet called `withdraw` by the time the window ends.
Location	- `FixedRatioCoinDisperser.sol:420`

Description	During pull redemptions, the contract increases `pendingWithdrawals[user]` and tracks the aggregate obligation in `pendingTotal`, both updated in `redeemPull` and paid out in `withdraw`.
	The owner-only `withdrawDust` correctly protects `pendingTotal`, sending only `balance - pendingTotal` to the owner after the window.
	
	However, the owner-only `rescueEth(address to, uint256 amount)` (allowed only after the window) does not account for `pendingTotal`. It only checks that `amount <= address(this).balance` and then transfers that ETH to an arbitrary address:
	`function rescueEth(address to, uint256 amount) external onlyOwner {`
	`        if (!(finalized && block.number > END_BLOCK)) revert RedemptionNotClosed();`
	`        if (to == address(0)) revert InvalidAddress();`
	`        if (amount == 0) revert AmountZero();`
	`    if (amount > address(this).balance) revert InsufficientEthBalance();`
	`        (bool success,) = to.call{value: amount}("");`
	`        if (!success) revert EthTransferFailed();`
	`}`
	
	This lets the owner (or anyone controlling the owner) empty the contract, including ETH that has already been earmarked for users' pull withdrawals. Users calling `withdraw` later will revert if there's no ETH left until the owner re-funds.

Mitigation	In `rescueEth`, enforce `amount <= address(this).balance - pendingTotal`. Alternatively, remove `rescueEth` and rely solely on `withdrawDust` for ETH extraction after the window, since `withdrawDust` already preserves `pendingTotal`.

rescueTokens can steal redeemed POINTS_TOKEN

ID	SAY-02
Status	Open
Risk	Medium
Business Impact	Owner can steal all points tokens that users have redeemed, potentially selling them or using them elsewhere if they have value.
Location	- `FixedRatioCoinDisperser.sol:404`

Description

The `rescueTokens` function is designed to rescue accidentally sent tokens, but it doesn't prevent the owner from rescuing the `POINTS_TOKEN` itself. All redeemed points are permanently locked in the contract, and the owner can steal them using this function.

```
function rescueTokens(address tokenAddr, address to, uint256 amount) external
onlyOwner {
       if (!(finalized && block.number > END_BLOCK)) revert
RedemptionNotClosed();
       if (to == address(0)) revert InvalidAddress();
       if (amount == 0) revert AmountZero();
       IERC20(tokenAddr).safeTransfer(to, amount);
   }
```

Exploitation scenario:

1. Users redeem 1000 `POINTS_TOKEN` during the window
2. These tokens are transferred to the contract and locked
3. After `END_BLOCK`, owner calls `rescueTokens(POINTS_TOKEN, ownerAddress, 1000)`
4. Owner receives all redeemed points tokens

Mitigation

We recommend preventing the rescuing of the points tokens.

```
if (tokenAddr == address(POINTS_TOKEN)) revert InvalidAddress();
```

Finalization ignores tracked `totalPayoutPool`

ID	SAY-03
Status	Open
Risk	Medium
Business Impact	Accounting discrepancies between the tracked pool and the actual pool.
Location	- `FixedRatioCoinDisperser.sol:175`

Description	The `finalize` function snapshots `address(this).balance` directly instead of using the tracked `totalPayoutPool` variable. This creates accounting inconsistencies and allows untracked ETH to be included in the payout calculation.
	
	`function finalize() external onlyOwner {`
	`        if (finalized) revert AlreadyFinalized();`
	
	`        uint256 pool = address(this).balance;`
	`        uint256 supply = POINTS_TOKEN.totalSupply();`
	`        if (pool == 0) revert PoolZero();`
	`        if (supply == 0) revert SupplyZero();`
	
	`        uint256 index = Math.mulDiv(pool, ONE, supply);`
	
	`        totalPayoutPool = pool;`
	`        totalRedeemablePoints = supply;`
	`        accPayoutPerPoint = index;`
	`        finalized = true;`
	
	`        emit Finalized(pool, supply, index);`
	`    }`

Mitigation	We recommend using the tracked `totalPayoutPool` variable in the finalize function.

EOA-only enforcement for push redemptions can be bypassed by contracts in construction

ID	SAY-04
Status	Open
Risk	Low
Business Impact	Bypasses intended policy and enables push payouts to contracts (minor risk because reentrancy is already guarded).
Location	- `FixedRatioCoinDisperser.sol:438`

Description	The `redeemPush` blocks contracts with:
	`function redeemPush(uint256 amount) external nonReentrant {`
	`if (!_isEoa(msg.sender)) revert ContractMustPull();`
	`[..]`
	
	`function _isEoa(address a) internal view returns (bool) {`
	`return a.code.length == 0;`
	`}`
	
	Addresses of contracts in their constructor have `code.length == 0`, so an attacker deploys a contract that calls `redeemPush` from its constructor, receives ETH, and still ends up with a contract address. This defeats the "EOA-only" intent.
	
	Reentrancy remains mitigated (`nonReentrant`) and the send uses `.call`, so the practical danger is limited, but the policy is enforceable only imperfectly.

Mitigation	If policy matters, accept a receiver parameter and apply checks on the originating EOA or use an allowlist/permit-based model. Alternatively, drop the EOA check entirely and rely on `nonReentrant` plus the pull model for contracts.

No slippage protection in redemptions

ID	SAY-05
Status	Open
Risk	Low
Business Impact	Users can receive significantly less value than expected, especially during high redemption periods or when the reserve is low.
Location	- `FixedRatioCoinDisperser.sol:226`

Description	Users cannot specify a minimum acceptable payout when redeeming points. If the reserve is depleted or manipulated between preview and execution, users may receive significantly less ETH than expected.
	
	`function redeemPull(uint256 amount) external nonReentrant {`
	`(uint256 payout, uint256 received) = _redeemCore(msg.sender, amount);`
	`pendingWithdrawals[msg.sender] += payout;`
	`pendingTotal += payout;`
	`emit Redeemed(msg.sender, amount, received, payout, false);`
	`}`
	
	Exploitation Scenario:
	
	1. User calls `previewPayout(1000)` and sees they'll get 10 ETH
	2. User submits transaction to `redeemPull(1000)`
	3. Before the transaction executes, the reserve is depleted by other users
	4. User's transaction executes but only receives 1 ETH due to line 309: `if (payout > reserve) payout = reserve;`
	5. User loses value with no recourse

Mitigation	We recommend adding a minimum payout parameter to the `redeemPull` function.

Preview and funding snapshot edge cases

ID	SAY-06
Status	Open
Risk	Low
Business Impact	Users may see a preview that doesn't match the exact payout they receive (no loss of funds).
Location	- `FixedRatioCoinDisperser.sol:175`

Description	The `finalize` snapshots using `address(this).balance` and `POINTS_TOKEN.totalSupply` to compute `accPayoutPerPoint`.
	
	After finalization, direct ETH sent via `receive` does not affect `totalPayoutPool` (used by `remainingReserve`), but it does affect the actual balance used by `_safeRemainingReserve` during redeem execution.
	
	The `previewPayout` caps by `remainingReserve` while execution caps by `_safeRemainingReserve`. If someone later sends ETH to the contract outside the pool accounting, the preview can under- or over-predict the real payout.
	
	`function finalize() external onlyOwner {`
	`if (finalized) revert AlreadyFinalized();`
	
	`uint256 pool = address(this).balance;`
	`uint256 supply = POINTS_TOKEN.totalSupply();`
	`if (pool == 0) revert PoolZero();`
	`if (supply == 0) revert SupplyZero();`
	
	`uint256 index = Math.mulDiv(pool, ONE, supply);`
	
	`totalPayoutPool = pool;`
	`totalRedeemablePoints = supply;`
	`accPayoutPerPoint = index;`
	`finalized = true;`
	
	`emit Finalized(pool, supply, index);`
	`}`

Mitigation	Expose a function that previews against `_safeRemainingReserve` for tighter estimates. Optionally clarify docs: `previewPayout` is an estimate - actual payout limited by on-chain actual reserve.

`moveBalanceToPool` is redundant

ID	SAY-07
Status	Open
Risk	Low
Business Impact	Confusion and operational errors.
Location	- `FixedRatioCoinDisperser.sol:153`

Description	The `moveBalanceToPool` increments `totalPayoutPool` up to `address(this).balance` before `finalize`. But `finalize` ignores `totalPayoutPool` and recomputes from `address(this).balance` anyway.
	
	The function emits a `Funded` event that may give the impression that something essential happened, but it does not affect the final rate.
	
	`function moveBalanceToPool() external onlyOwner {`
	`        if (finalized) revert AlreadyFinalized();`
	`        uint256 balance = address(this).balance;`
	`        if (balance == 0) revert PoolZero();`
	
	`        // Only add the difference to avoid double-counting`
	`        uint256 currentPool = totalPayoutPool;`
	`        if (balance > currentPool) {`
	`            uint256 difference = balance - currentPool;`
	`            totalPayoutPool += difference;`
	`            emit Funded(msg.sender, difference);`
	`        }`
	`    }`

Mitigation	Drop `moveBalanceToPool` or clearly document it as cosmetic bookkeeping pre-finalization to avoid integration mistakes.

Owner can be changed to a malicious address

ID	SAY-08
Status	Open
Risk	Low
Business Impact	Immediate owner change allows a malicious actor to pause or drain funds
Location	- `FixedRatioCoinDisperser.sol:386`

Description	The `transferOwnership` function has no two-step transfer. If the owner's private key is compromised or the owner makes a mistake, a malicious actor can immediately take control, or the protocol becomes locked forever..
	
	 `function transferOwnership(address newOwner) external onlyOwner {`
	`        if (newOwner == address(0)) revert OwnerZeroAddress();`
	`        address prev = owner;`
	`        owner = newOwner;`
	`        emit OwnershipTransferred(prev, newOwner);`
	`    }`

Mitigation	We recommend implementing a two-step ownership transfer.

Events exist for withdrawals and dust, but no event for `rescueEth` and `rescueTokens`

ID	SAY-09
Status	Open
Risk	Low
Business Impact	Forensics and monitoring challenge, as incident response is harder.
Location	- `FixedRatioCoinDisperser.sol:420`

Description	The `rescueEth` and `rescueTokens` do not emit events. Given their power (especially ETH rescue), lack of an event hampers monitoring.
	
	`function rescueEth(address to, uint256 amount) external onlyOwner {`
	`       if (!(finalized && block.number > END_BLOCK)) revert`
	`RedemptionNotClosed();`
	`       if (to == address(0)) revert InvalidAddress();`
	`       if (amount == 0) revert AmountZero();`
	`       if (amount > address(this).balance) revert InsufficientEthBalance();`
	
	`       (bool success,) = to.call{value: amount}("");`
	`       if (!success) revert EthTransferFailed();`
	`}`

Mitigation	Implement event emission for `rescueEth` and `rescueTokens`.

Errors are defined, but never used in the logic

ID	SAY-10
Status	Open
Risk	Informational
Business Impact	Risk of some missing functionality, redundant code.
Location	- `FixedRatioCoinDisperser.sol:28`

Description	The `FixedRatioCoinDisperser` contract implements multiple custom errors, of which a few are never used in the logic. These are:
	• `NoExcess`
	• `WithdrawFailed`
	• `InsufficientBalance`
	
	`// --- Custom Errors ---`
	`error TokenZeroAddress();`
	`error OwnerZeroAddress();`
	`error InvalidBlockRange();`
	`error NotOwner();`
	`error AlreadyFinalized();`
	`error PoolZero();`
	`error SupplyZero();`
	`error NotFinalized();`
	`error NoExcess();`
	`error WindowActive();`
	`error NoDust();`
	`error WithdrawFailed();`
	`error ContractMustPull();`
	`error NothingToWithdraw();`
	`error ContractPaused();`
	`error OutOfWindow();`
	`error AmountZero();`
	`error SnapshotZero();`
	`error NoPointsLeft();`
	`error OverCapacity();`
	`error BalanceManipulation();`
	`error PayoutZero();`
	`error RedemptionNotClosed();`
	`error InvalidAddress();`
	`error InsufficientBalance();`
	`error InsufficientEthBalance();`
	`error EthTransferFailed();`

Mitigation	We recommend checking if some logic related to these errors was not missed or removing them from the codebase.

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