Smart Contract Audit Report for Rise

Testers

1. Or Duan
2. Avigdor Sason Cohen

Table of Contents

Management Summary

Rise contacted Sayfer to perform a security audit on their smart contracts in December 2024.

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 Rise’s smart contracts.

Over the research period of 4 weeks, we discovered 11 vulnerabilities in the contracts.

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 Rise 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.

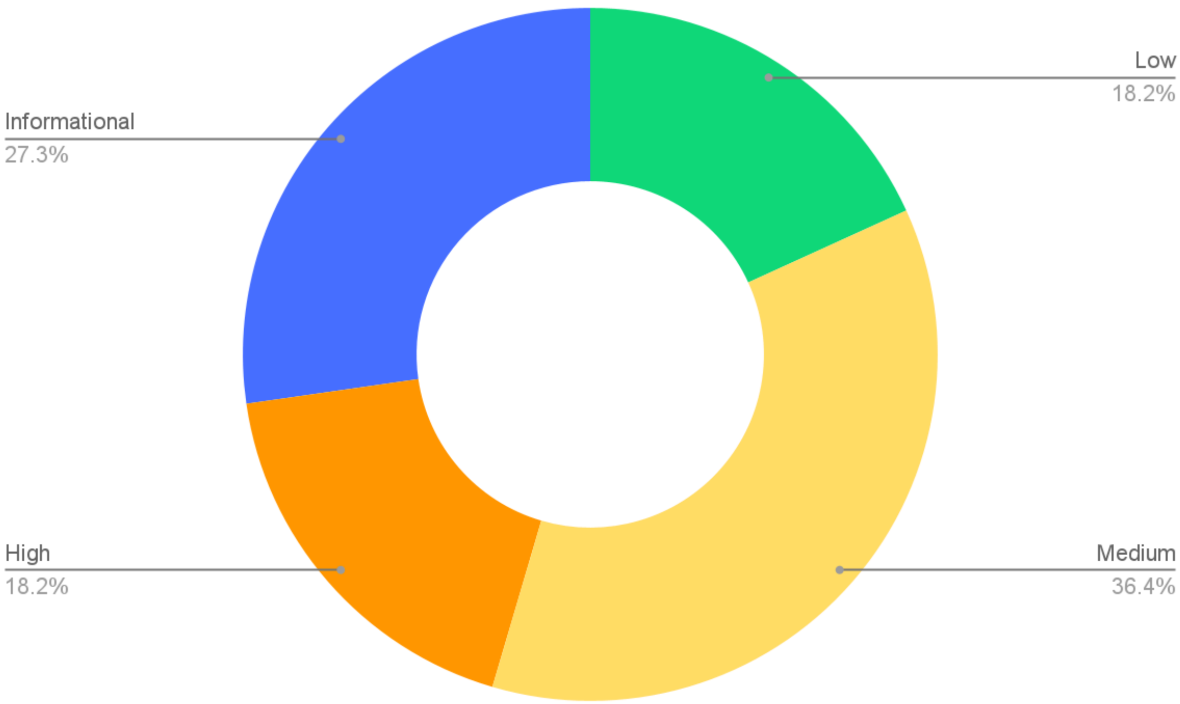
By combining these two factors, we can create a comprehensive understanding of the risk posed by a particular issue and provide our clients with a clear and actionable assessment of the severity of the issue. This approach allows us to prioritize our recommendations and ensure that our clients receive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

Risk is defined as follows:

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	2	4	2	0	3

Approach

Introduction

Rise 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 contracts as the scope of the project.

Contract	SHA-256
`RiseAccess.sol`	`e54a351809fa2afa99185ca3132f1b43ad6622f6860556c9fff101e8e87dcf7f`
`RiseAccount.sol`	`10cfb98299d8349d673c46db62111518461f67cec2eac0e445fb95b597bdb772`
`RiseAccountSubscriptionUsage.sol`	`978f168998050a04ed5ead613d764dae4e425c91e9add960a4a45f55d0890fad`
`RiseDeployFactory.sol`	`3789dd3909c4b6d5ff60b627d62384e88b84c079e78765b963e0ec6bec7988ae`
`RiseForwarder.sol`	`c4f67c52c921a79f9e7110a57447f45c3ce83beac48203eceec73ecaab59e725`
`RiseID.sol`	`34f67695d68ccee7418631fb43522eaf28648c6ef501795696c71dce688c52b8`
`RisePaymentHandler.sol`	`27cde2d7dbdb361b01f6928ccf1116510d0303fbbed534eda10ecf5e8a923ddd`
`RisePayToken.sol`	`18304529023fb65fe20788f3eaeab2097418ecc7b360cec9f67299145b41bb04`
`RiseProxy.sol`	`dc3fbbf2484b3c476df5844c6c3a96a9a9cf429d3c4104ed4371ed68b59c8976`
`RiseRouter.sol`	`0fd5369ba7297f8d2a5fc4f117bcc271db50bb9e2efbaf14ebb2ccb88de1a147`
`Governors/RiseAccessGovernor.sol`	`1a6618aa3651a14f6554e7e26b3c1b4e185d4bba9b5dcc1fb720a8963e8819ac`
`Governors/RiseAccountGovernor.sol`	`6c7d6ab53a801d141291e92e10a763859358a1daf275ad5393368a918894d271`
`Governors/RiseDeployFactoryGovernor.sol`	`cd19811df3133c59b00598192f3ac37cc2b03e50509cbeef41d488159178b090`
`Governors/RiseDepositGovernor.sol`	`2adeedb536e22b40fe394d56a0ff9d2a6fdfc898d714a67f927bb82c6f0cc9e1`
`Governors/RisePaymentHandlerGovernor.sol`	`476643c92bee2fab337a0b7f3d1ba608840bbc0af6d7b838b7f20c9b1a5c155e`
`Governors/RiseTokenGovernor.sol`	`2d1d4988439ec81a45cbb1bb07d53c2eff8e7674cb275b31cf98475966b088e5`
`Ramps/RiseRampDeposit.sol`	`3cc63d3ed148584ade0059a03690b178fdba5756daa46a09dfaa9d339be45b52`

Ramps/RiseRampDepositCCIP.sol	9502d05ecb38b075b987cb3a63d829e17ebe7bbd4e37b38eb89f1e85bb04ca80
Ramps/RiseRampERC20Token.sol	a3af5f5f7200d14cac0212b7ac9305b2a967d7821e8bdda2d6c4dff31c675d79
Ramps/RiseRampUSDC_PYUSD.sol	aa5e3ce3e045aec2f5c7fcadb3ccc8348ac6b18ccf59af26b4285b868b8e1347
Ramps/RiseRampUSDCDAI.sol	27baa3366ea4eeec2dfba0f9478ab913b3d1ff5897ad93184a1f1e8862aa26e8
Ramps/RiseRampWithdrawCCIP.sol	d2a60ec07d0c3a899c89245c426025540917649d02eb14a566cda0244323cd4c
Ramps/RiseRampWithdrawERC20Token.sol	6f4e945b10773dd5ad5f4f5a81aaef2d879053dc09c9bd1ad87ec3f5846d44de
Ramps/RiseRampWithdrawExchange.sol	9c422ae53bb91e009df535d7fda7fe347638d4775caaddda60ec847fc51a4c0f
Ramps/RiseRampWithdrawInternationalUSD.sol	a756934cb9034d8f6fd7f4cc7ca51ed1394432ccb0fba887195d91895e2561df
Ramps/RiseRampWithdrawInternationalUSDManual.sol	e5e69da3fd9b23d81bb0db0931016db82f328ea6f57a9566497d17784c064e6d
Ramps/RiseRampWithdrawSwap.sol	b26e2b0ac8e684d82a4a4d9a8d92f67fac7aac279e1e4fc71b5a8ee2724e9beb
Ramps/RiseRampWithdrawUnblock.sol	4f5641f52654abe66c65865520b1c6740205e4219ce006087db2d728686eed38
RiseRampWithdrawUSDUS.sol	190c1b76d485b35006e8cc581a92946bd506dfcb503186d372f4967bba1e3557

Our tests were performed from 05/12/2024 to 26/12/2024.

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

`RisePayToken.executeTransfers` Only Works for `idx = 0`

ID	SAY-01
Status	Fixed
Risk	High
Business Impact	The function `executeTransfers` reverts whenever `idx` is non-zero, which means that some intended partial transfers will not be possible and all transfers have to be executed in order.
Location	- `RisePayToken.sol; executeTransfers(uint256, uint256)`

Description

`executeTransfers(uint256, uint256)` calls `pendingTransferHashesSlice(uint256, uint256)` with the `idx` and `count` argument and then iterates over the result:

- `RisePayToken.sol:109-116`

```
bytes32[] memory transferHashes = this.pendingTransferHashesSlice(idx, count);
for (uint256 i = 0; i < count; i++) {
   bytes32 _hash = transferHashes[idx + i];
   address _account = $.transferRegistry[_hash];
   uint256 _amount = $.transferLedger[_hash];
   $.pendingTransfers.remove(_hash);
   _mint(_account, _amount);
}
```

In the iteration, the elements of transferHashes are accessed via the index `idx + i`, i.e. `idx` is added as a fixed constant to the loop iterator `i`. However, `pendingTransferHashesSlice(uint256, uint256)` returns an array of size `count` that starts at index 0:

- `RisePayToken.sol:150-153`

```
_transfers = new bytes32[](count);
for (uint256 i = 0; i < count; i++) {
   _transfers= $.pendingTransfers.at(idx + i);
}
```

	This means that when `idx` is non-zero, an out-of-bounds array access will be attempted, causing a revert.
Mitigation	Access the array elements at `i` instead of `idx + i`.

Blacklist Can Be Bypassed via Role Renouncement

<table>
<tr><td>ID</td><td>SAY-02</td></tr>
<tr><td>Status</td><td>Fixed</td></tr>
<tr><td>Risk</td><td>High</td></tr>
<tr><td>Business Impact</td><td>Users who have been blacklisted can bypass the restriction by renouncing their blacklist role, effectively removing themselves from the blacklist. This completely undermines the blacklisting mechanism, which is critical for regulatory compliance and security measures.</td></tr>
<tr><td>Location</td><td>
- `RisePayToken.sol; internalTransfer(address, bytes32, uint256)`
- `And all other functions using RISE_IS_ADDRESSES_BLOCKED...`
</td></tr>
<tr><td>Description</td><td>
The project implements blacklisting through OpenZeppelin's `AccessControl` by assigning the `RISE_IS_ADDRESSES_BLOCKED` role to blacklisted addresses.

- For example, `RisePayToken.sol:89-90`

```
if (hasRole(RiseGlobals.RISE_IS_ADDRESSES_BLOCKED, _msgSender())) revert RiseGlobals.Rise_Unauthorized(_msgSender());
if (hasRole(RiseGlobals.RISE_IS_ADDRESSES_BLOCKED, recipient)) revert RiseGlobals.Rise_Unauthorized(recipient);
```

However, due to the inherited `renounceRole(bytes32, address)` function from `AccessControl`, users can simply remove themselves from the blacklist.

- `AccessControl.sol:157-163`

```
function renounceRole(bytes32 role, address callerConfirmation) public virtual {
    if (callerConfirmation ≠ _msgSender()) {
        revert AccessControlBadConfirmation();
    }

    _revokeRole(role, callerConfirmation);
}
```
</td></tr>
<tr><td>Mitigation</td><td>Consider implementing a custom blacklist mechanism instead of using `AccessControl` roles. An alternative approach may involve overriding and modifying `renounceRole(bytes32, address)` to prevent renouncement of the blacklist role.</td></tr>
</table>

Inconsistencies in Blacklist Checks

ID	SAY-03
Status	Fixed
Risk	Medium
Business Impact	The blacklist is not enforced in all functions, allowing transactions that should be blocked to go through.

Location	- `RiseAccount.sol`
	  - `sendEther(RiseRequests.RiseEtherTransferRequest, calldata)`
	  - `tokenTransfer(RiseRequests.RiseEtherTransferRequest, calldata)`
	- `RisePaymentHandler.sol`
	  - `_processEtherRule(RiseRequests.RisePaymentHandlerConfig, uint256)`

Description	`RiseAccount` has multiple functions to transfer funds and manage approvals, namely `sendEther`, `tokenTransfer`, and `setTokenTransferApproval`.
	
	`setTokenTransferApproval(RiseRequests.RiseEtherTransferRequest, calldata)` calls the modifier `isValidToken(address, bool)`, which (among other things) checks if the token is blocked (i.e. has the role `RISE_IS_ADDRESSES_BLOCKED`).
	• `RiseAccount.sol:132`
	`function setTokenTransferApproval(RiseRequests.RiseTokenApprovalRequest calldata req) external isValidToken(req.token, false) onlyTreasurer`
	
	But this is not done in the similarly important `tokenTransfer( ... )` nor in `sendEther( ... )`.
	• `RiseAccount.sol:118`
	`function tokenTransfer(RiseRequests.RiseTokenTransferRequest calldata req) external nonReentrant onlyTreasurer`
	• `RiseAccount.sol:108`
	`sendEther(RiseRequests.RiseEtherTransferRequest calldata req) external nonReentrant onlyTreasurer`

	Moreover, no function checks if the recipient is a blocked address, which other functions in other contracts (e.g. `RisePaymentHandler.setTransferRules( … )`) do.
	• `RiseRequests:243`
	`if (hasRole(RiseGlobals.RISE_IS_ADDRESSES_BLOCKED, config.destination))`
	`revert RiseGlobals.Rise_Unauthorized(config.destination);`
	
	Even within `RisePaymentHandler`, these inconsistencies can also be observed. For token transfers (in `_processTokenRule( … )`), the checks are done, while they are missing for native token transfers (in `_processEtherRule( … )`).

Mitigation	Always check whether the token (unless it is the native token), the recipient, or the destination are on the blacklist.

Fixed Exchange Rate Can Lead to Under– or Overpayment

ID	SAY-04
Status	Acknowledged
Risk	Medium
Business Impact	If the LINK/USDC price changes quickly and the owner fails to update the exchange rate, the user may be charged too much or too little fees for deposits.
Location	- `RiseRampDepositCCIP.sol; execute(address, address, bytes32, bytes)` - `RiseRampDepositCCIP.sol; setLinkToUSDC(uint32)`

Description	`RiseRampDepositCCIP` multiplies the `fees` value (in LINK) by the exchange rate `$.linkToUSDC`, which is configurable by the owner via `setLinkToUSDC(uint32)`.
	● `RiseRampDepositCCIP:96`
	`uint256 usdcFee = $.linkToUSDC * fees;`
	● `RiseRampDepositCCIP:63-67`
	`function setLinkToUSDC(uint32 _linkToUSDC) external onlyAdmin {`
	`RiseRampCCIPStorageVars storage $ = _getVars();`
	`if (_linkToUSDC == 0) revert RiseGlobals.Rise_InvalidRequestWithReason("RiseRampCCIP: Link to USDC must be greater than 0");`
	`$.linkToUSDC = _linkToUSDC;`
	`}`
	
	This is not recommended, as the LINK/USDC price can fluctuate heavily and the owner may not be able to always update it in a timely manner. Moreover, fetching prices off-chain can also be error-prone because of things like API issues or inconsistent prices between centralized / decentralized exchanges. With the current design, an always-online bot network that aggregates multiple price sources would be required, i.e. you would need to build your own oracle.

Mitigation	It is recommended to use a common oracle solution (like PYTH or Chainlink) to set exchange rates.

A 1:1 Exchange Rate for DAI/USDC Can Lead to Losses

ID	SAY-05
Status	Fixed
Risk	Medium
Business Impact	The contract assumes an exchange rate of 1:1 for DAI/USDC, which may not always hold. This could be abused if one of the tokens depegs.

Location	- `RiseRampUSDCDAI.sol`
	    - `DAI_to_USDC(address, address, address)`
	    - `USDC_to_DAI(address, address, address)`

Description	`RiseRampUSDCDAI` assumes that one DAI is worth one USDC. While this is usually true, there can be events when it does not hold. USDC for instance depegged in the past, which would have opened an arbitrage opportunity (at the expense of Rise).
Mitigation	It is recommended to query the prices with an oracle.

Unbounded Loop

ID	SAY-06
Status	Fixed
Risk	Medium
Business Impact	The function could become unusable if the `tokenTransferConfigs` array grows too large and requires an excessive amount of gas for loading the entire array into memory, causing reverts.
Location	- `RisePaymentHandler.sol; processTokenTransfers(address)`

Description	`processTokenTransfers(address)` has an unbounded loop that could make it vulnerable to denial of service.
	• RisePaymentHandler.sol:141-145
	`for (uint256 i = 0; i < configs.length; i++) {`
	`uint256 amountToTransfer = _calculateTransferAmount(token, configs[i], startingBalance, lastAmountTransferred);`
	`lastAmountTransferred = _processTokenRule(token, configs[i], amountToTransfer);`
	`totalTransferred += lastAmountTransferred;`
	`}`
	
	Since the storage array is loaded entirely into memory and multiple external calls are performed in the loop, the gas costs can quickly exceed block gas limits if the array grows large enough, with no way to partially process the transfers.

Mitigation	Consider implementing pagination to process transfers in smaller batches, allowing for partial processing across multiple transactions. Alternatively, a pull pattern could be implemented where recipients claim their tokens individually rather than having them pushed in a single transaction.

Inflexible Access Control System

ID	SAY-07
Status	Fixed
Risk	Low
Business Impact	Each role admin also has to be a global admin because the grant / revoke functions are gated to that role.

Location	- `RiseAccessGovernor.sol`
	  - `batchRoleGrantOrRevoke(string, bool, address[])`
	  - `multiRoleGrantOrRevoke(address, bool, bytes32[])`

Description

The two specified functions have the `onlyAdmin` modifier which means that they are only callable by the global admin.

- `RiseAccessGovernor.sol:33-38`

```
modifier onlyAdmin() {
   if (!hasRole(RiseGlobals.RISE_IS_ACCESS_ADMIN, msg.sender)) {
       revert
RiseGlobals.Rise_UnauthorizedRole(RiseGlobals.RISE_IS_ACCESS_ADMIN,
msg.sender);
   }
   _;
}
```

It is not clear if this is intended. If so, it means that the role system is very inflexible in practice. Because of the `onlyAdmin` modifier, only an address that is also the global admin (and can therefore set itself as role admin anyways) can be a role admin and no separation of concerns (with multiple addresses) is possible.

Curiously, the functions also also check whether the `msg.sender` is a role admin of the requested role.

- `RiseAccessGovernor.sol:127`

```
if (!hasRole(access.getRoleAdmin(role), msg.sender)) revert
RiseGlobals.Rise_UnauthorizedRole(access.getRoleAdmin(role),
msg.sender);
```

- `RiseAccessGovernor.sol:148`

```
if (!hasRole(access.getRoleAdmin(roles[i]), msg.sender)) revert
RiseGlobals.Rise_UnauthorizedRole(access.getRoleAdmin(roles[i]),
msg.sender);
```

Mitigation	It is recommended to allow role admins to call this function and not require the caller to be a global admin.

Insufficient Array Length Validation in Batch Execution

ID	SAY-08
Status	Fixed
Risk	Low
Business Impact	The current array length validation could allow batch operations with mismatched array lengths, potentially leading to unexpected behavior or transaction failures.
Location	- `RiseERC725X.sol; _executeBatch(uint256[], address[], uint256[], bytes[])`

Description	`_executeBatch( ... )` performs batch operations using multiple arrays as parameters. However, the current length validation logic doesn't properly ensure all arrays have equal length due to incorrect usage of OR operators.
	● `RiseERC725X.sol:140-145`
	`if (`
	`  operationsType.length != targets.length ||`
	`  (targets.length != values.length || values.length != datas.length)`
	`) {`
	`  revert RiseGlobals.Rise_InvalidRequest();`
	`}`
	
	The current validation can pass even when array lengths are mismatched. For example:
	● If operationsType.length equals targets.length
	● AND values.length equals datas.length
	● BUT operationsType.length differs from values.length
	This could lead to out-of-bounds errors during execution or incomplete batch processing.

Mitigation	Replace the current validation with AND (&&) operators to ensure all arrays have the same length.

Unimplemented USDC / PYUSD Conversion

ID	SAY-09
Status	Acknowledged
Risk	Informational
Business Impact	The PYUSD / USDC conversion (and vice-versa) is left unimplemented.

Location	- `RiseRampUSDC_PYUSD.sol;`
	    - `PYUSD_to_USDC(address, address)`
	    - `USDC_to_PYUSD(address, address)`

Description	The two specified functions contain a comment «`// TODO: uniswap?`» and the conversion was left unimplemented.
Mitigation	It is recommended to address all open TODOs and implement the conversion.

Unnecessary Wrappers around ERC20 Functions

ID	SAY-10
Status	Acknowledged
Risk	Informational
Business Impact	The codebase's maintainability is reduced by implementing unnecessary wrapper functions that mimic standard ERC20 functions while adding little value. This increases the surface area for potential bugs and makes the code harder to audit and maintain.

Location	- `RiseID.sol`
	    - `transfer(RiseRequests.Transfer)`
	    - `transferFrom(RiseRequests.TransferFrom)`
	    - `approve(RiseRequests.Approve)`
	    - `approve(RiseRequests.ApprovalChange)`

Description	`RiseID` implements multiple wrapper functions around standard ERC20 operations that add minimal validation while making the code more complex.
Mitigation	Review these functions and consider removing them.

Gas Optimizations

ID	SAY-11
Status	Fixed
Risk	Informational
Business Impact	Gas usage could be decreased.
Location	- `RisePaymentHandler.sol:116` - `RiseRouter.sol:99`
Description	In two places, the gas usage could be decreased by removing unnecessary operations: • `RisePaymentHandler.sol:116` : The check `config.ramp != address(0x0)` is performed twice. The second one could be removed. • `RiseRouter.sol:99` : `$.routes[ref]` is read twice from storage. Once for the zero-address check and the second time when returning it. It is recommended to store it in a temporary variable.
Mitigation	Consider implementing these optimizations.

SAYFER

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