Smart Contract Audit Report for Station

Testers

1. Or Duan
2. Avigdor Sason Cohen

Table of Contents

Management Summary

Station 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 Station's smart contracts.

Over the research period of 3 weeks, we discovered 10 vulnerabilities in the contracts.

Overall, the system security posture is good, and we believe that after fixing our findings, it will be ready for mass production.

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

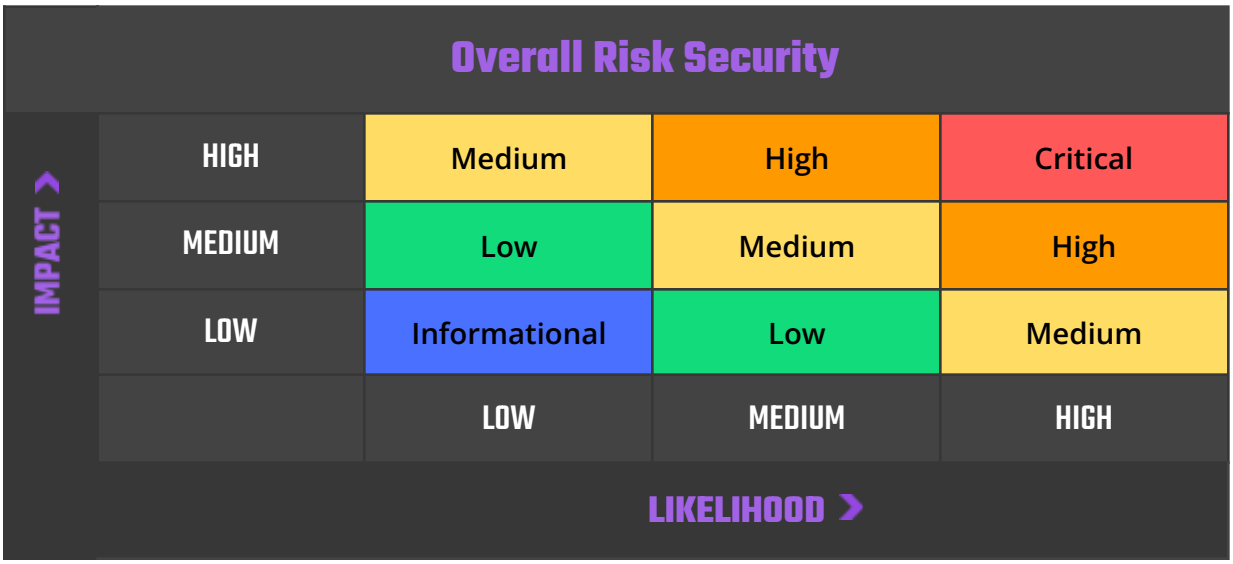
Risk Methodology

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.

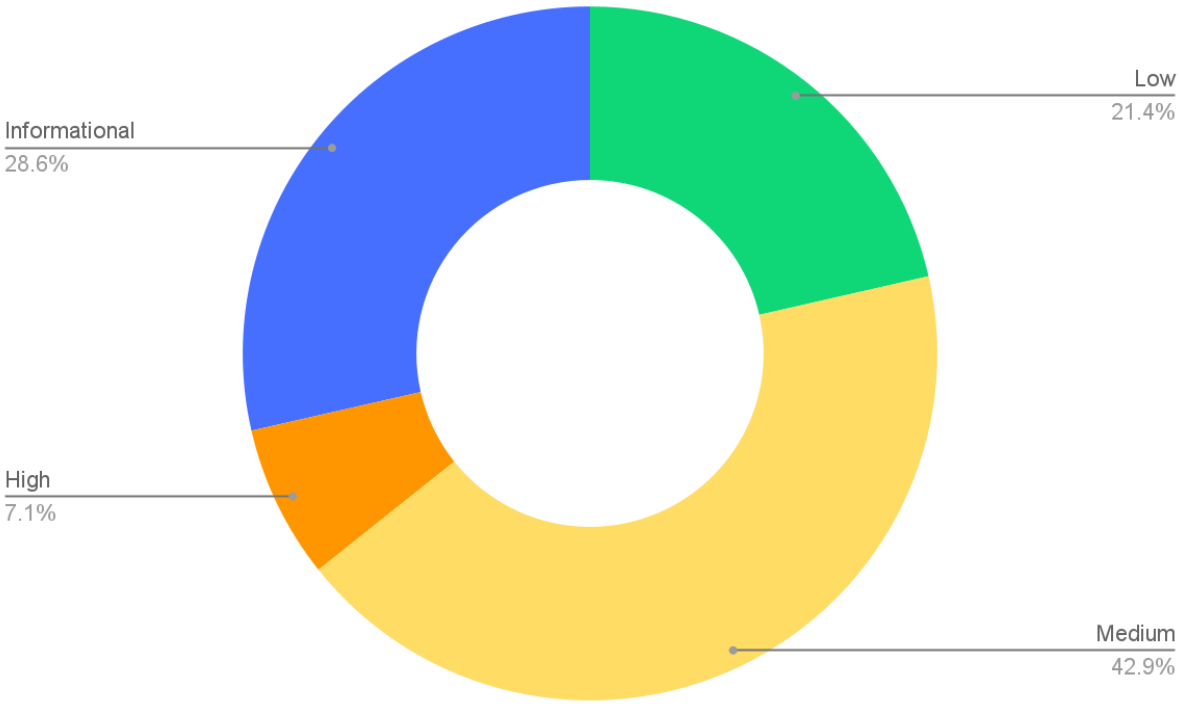
Our risk assessment model is based on two key factors: **IMPACT** and **LIKELIHOOD**. Impact refers to the potential harm that could result from an issue, such as financial loss, reputational damage, or a non-operational system. Likelihood refers to the probability that an issue will occur, taking into 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	3	6	1	0	4

Approach

Introduction

Station contacted Sayfer to perform a security audit on their smart contracts.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for the aforementioned contracts.

Scope Overview

Together with the client team we defined the following contract as the scope of the project.

- 0xrails
  - 9e0c98dfcfee6c5f04219ec90ed6bdda61b75ff2
- GroupOS
  - 1.1.0
- GroupOS Safe
  - 8fe4a2bdd16e1dcf067d1750899edd623c92618c

Our tests were performed between October to November 2023.

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).

Audit Findings

Lack of Emergency Pause

ID	SAY-01
Status	Acknowledged
Risk	Medium
Business Impact	No possibility to pause contracts in the event of an emergency.
Location	—
Description	The whole solution does not implement any kind of emergency stop pattern. Lack of such a mechanism might be essential, e.g. during attacks. For instance, without such a mechanism the minting in *`FreeMintController`*, *`GasCoinPurchaseController`*, *`StablecoinPurchaseController`* can not be temporarily disabled.
Mitigation	It is recommended to implement an emergency-stop pattern, such as OpenZepplin's *`Pausable`* and enforce users to manage it from a multisig wallet

0xrails

Time Complexity of ERC721A Implementation is Linear

ID	SAY-0x-01
Status	Fixed
Risk	Medium
Business Impact	For very large batch sizes, some operations may revert.
Location	- `ERC721Internal.sol; _batchMarkerDataOf(uint256)`
Description	When *`_batchMarkerDataOf()`* is called for a token ID that sits directly at the end of a batch, it has to traverse the whole batch to reach the index that stores the owner. This traversal is relatively expensive because it involves a storage read in every loop iteration. In the worst case (for very large batch sizes), it may even be more expensive than the block gas limit, leading to non-transferrable tokens.
Mitigation	A simple solution would be to enforce an upper batch size limit by either enforcing. An upper limit for the quantity when minting or splitting a mint into multiple batches when it is larger than a certain size. Another approach would be to redesign the data structures in order to make traversal constant. One way to do this would be to insert “jump marker” every n (for instance with n=20) elements in a batch. These would store the index of the batch start. However, this approach also has some drawbacks. Minting would technically no longer be in constant time (it would be linear, but only with a slope of 1/n) and burning/transfers would be more involved.

Prefunding Entry Points Doesn't Always Work

ID	SAY-0x-02
Status	Fixed
Risk	Medium
Business Impact	The function *`preFundEntryPoint()`* will not work and revert when a user calls it with ETH attached.
Location	- `src/cores/account/BaseAccount.sol:32; preFundEntryPoint()`
Description	*`preFundEntryPoint()`* is a payable function that's supposed to transfer the complete balance of the *`BaseAccount`* contract plus everything that was sent along the call to the entry point contract. To do so, it adds msg.value to *`address(this).balance`*: `uint256 totalFunds = msg.value + address(this).balance;` However, the value of *`address(this).balance`* already includes *`msg.value`*, i.e. the funds that were sent with this call. Therefore, *`totalFunds`* will be greater than the ETH that is in the contract (if msg.value is non-zero), causing the transfer to revert.
Mitigation	Set *`totalFunds`* to *`address(this).balance`*.

Accidental Ownership Renunciation

ID	SAY-0x-03
Status	Fixed
Risk	Medium
Business Impact	Accidentally calling this function will lead to permanent loss of ownership.
Location	- `access/ownable/Ownable.sol:16`
Description	Renounce ownership might not be the best fit for all projects since many of them do plan to keep the centralization of ownership. The solution implements both OpenZepplin's *`Ownable`* and 0xrails's *`Ownable`*. In both cases access-management contracts implement the *`renounceOwnership()`* function. This function can be mistakenly called, causing accidental loss of ownership.
Mitigation	Consider using OwnableTwoStep approach that requires 2 steps before execution transferring(or in this case, renunciation). Or if centralization should always exist, removing the *`renounceOwnership()`* function entirely.

GroupOS

Stablecoin Purchase Controller Assumes a 1:1 Peg for all Tokens

ID	SAY-Gr-01
Status	Fixed
Risk	High
Business Impact	A depeg of a stablecoin can result in significant discounts and might open arbitrage opportunities.
Location	- `src/membership/modules/StablecoinPurchaseController.sol; mintPriceToStablecoinAmount(uint256, address)`
Description	The function *`mintPriceToStablecoinAmount()`* converts the mint price of a collection to the correct amount in the queried stablecoin. However, it only converts the decimals of the price to do so. This implicitly assumes that the value of the queried stablecoin is exactly 1 (USD, EUR, ETH, or whatever the configured currency is). The price of many stablecoins has fluctuated historically and there were depeg events with large price drops in the past. In such a scenario, a user can buy this coin cheaply to get a discount for the collection. If the collection is also traded on a secondary market (with a similar price to the mint price), this is also an arbitrage opportunity and would most likely result in a price drop on the secondary market.
Mitigation	Consider incorporating price oracles into the system. This would enable it to adjust prices to the current market price of the coin, therefore eliminating any discounts or arbitrage opportunities.

Ownable Used Instead of Ownable2Step

ID	SAY-Gr-02
Status	Fixed
Risk	Medium
Business Impact	A mistaken transfer can lead to potential loss of ownership.
Location	- `src/membership/modules/FeeController.sol`
Description	*`FeeController`* implements OpenZepplin's *`Ownable`* solution. The *`Ownable`* provides single step ownership transfer which is prone to mistaken transfer ownership role to invalid EOA. On the contrary, the *`TokenFactory`* implements 0xrails's *`Ownable`*, which implements two-step ownership transfer.
Mitigation	It is recommended to implement *`Ownable2Step`* solution in every case.

Factories Can Be Used to Create Tokens with Arbitrary Implementation

ID	SAY-Gr-03
Status	Fixed
Risk	Medium
Business Impact	Attackers might trick users into thinking malicious tokens are valid GroupOS tokens.
Location	– TokenFactory.sol
Description	When a user creates a token via createERC20, createERC721, or createERC1155, they provide the address of the implementation. This address can be arbitrary, it does not have to be an implementation that was created or vouched by Station. While this is not necessarily a problem, attackers might abuse the factories for the deployment of malicious tokens. They can then claim and advertise that these tokens are valid, secure tokens that were created with the GroupOS toolkit by showing that they were created by the factory. This may lead to reputational damage for the company.
Mitigation	Consider implementing a whitelist for the implementation addresses.

Empty Transfers in FreeMintController

ID	SAY-Gr-04
Status	Fixed
Risk	Low
Business Impact	Not only does the empty transfer use up unnecessary gas, but some ERC20 tokens may actually revert zero-value transfers.
Location	- `src/membership/modules/FreeMintController.sol:109; _batchMint(address, address, uint256)` - `src/lib/FeeController.sol:120; _collectFeeAndForwardCollectionRevenue(address, address, address, address, uint256, uint256)`
Description	*`_batchMint(address, address, uint256)`* in *`FreeMintController`* is meant to mint NFT tokens without providing any payment to the *`payoutAddress`*, as the *`unitPrice`* is set to 0. Only an applicable fee from *`FeeManager`* is applied. But it was found that it performs additional ether transfer or ERC20 tokens transfer with 0 amount to the *`payoutAddress`* in every case within *`_collectFeeAndForwardCollectionRevenue`*.

- Notice how the last variable in batch mint *`unitPrice`* is set to null in the call to *`_collectFeeAndForwardCollectionRevenue`*:

```
function _batchMint(address collection, address recipient, uint256 quantity)
    internal
    usePermits(_encodePermitContext(collection))
{
    require(quantity > 0, "ZERO_AMOUNT");

        _collectFeeAndForwardCollectionRevenue(collection, address(0), address(0), recipient, quantity, 0);

    IERC721Rails(collection).mintTo(recipient, quantity);
}
```

- But in *`_collectFeeAndForwardCollectionRevenue`*, that 0 is multiplied with *`quantity`* to get the value for transfer.

```
(bool success,) = payoutAddress.call{value: quantity * unitPrice}("");
```

Mitigation

The transfer in *_collectFeeAndForwardCollectionRevenue* should be done like this:

```
if (unitPrice==0) {
   (bool success,) = payoutAddress.call{value: quantity}("");
}
else {
        (bool success,) = payoutAddress.call{value: quantity * unitPrice}("");
}
```

Mint Functions Marked as Payable

ID	SAY-Gr-04
Status	Fixed
Risk	Low
Business Impact	Possibility of mistakenly sent ether being lost.
Location	- `src/membership/modules/StablecoinPurchaseController.sol:250-272`

Description	*`StablecoinPurchaseController`* allows users to mint some NFT tokens in exchange of payment done in stable coins. However, this particular controller has mint functions marked as *`payable`*, thus it allows to attach ether to the function calls. In rare cases, it could be possible to trigger these functions with ether attached, where the actual payment would be done in a stable coin.
	
	`function mint(address collection, address paymentCoin) external payable`
	`{`
	`  _batchMint(collection, paymentCoin, msg.sender, 1);`
	`}`
	
	Also, it appears that the *`GasCoinPurchaseController`* contract is meant to receive payment in ether.

Mitigation	We recommend removing the *`payable`* keyword from the relevant functions.

Unnecessary Usage of a *payable* Address in *withdrawFees(address[])*

ID	SAY-Gr-05
Status	Fixed
Risk	Informational
Business Impact	Code clarity
Location	- src/lib/FeeController.sol:81; withdrawFees(address[])
Description	*withdrawFees(address[])* allows the solution's owner to withdraw ether or ERC20 tokens. To withdraw ether, the low-level *call()* function is used. The *call()* is called on a *payable* address, instead of a normal *address* variable. `(bool success,) = payable(recipient).call{value: amount}("");` The payable keyword is applicable to the functions that process the ether or to addresses that want to use *transfer()* function. Thus, in this context, a payable address is not needed.
Mitigation	You can simply write: `(bool success,) = address(recipient).call{value: amount}("");`

withdrawFees(address[]) uses ERC20's transfer instead of safeTransfer

ID	SAY-Gr-06
Status	Fixed
Risk	Informational
Business Impact	While usage of *transfer()* instead of *safeTransfer()* is considered a deviation from leading security practices, we identified no concrete threat in this context, so this finding is marked as informational.
Location	- src/lib/FeeController.sol:85; withdrawFees(address[])
Description	The *withdrawFees()* function allows the solution's owner to withdraw ether or ERC20 tokens. To withdraw the ERC20 tokens the *transfer()* function is used. `IERC20Metadata(paymentTokens[i]).transfer(recipient, amount);`
Mitigation	Consider using *safeTransfer()* instead of *transfer()* as a matter of habit.

i++ used instead of *++i*

ID	SAY-Gr-07
Status	Fixed
Risk	Informational
Business Impact	An often ignored solidity fact is that *++i* consumes less gas inside loops compared to *i++*.
Location	—
Description	We noticed that there's a tendency in the codebase to use the less efficient *i++* inside loops, rather than its recommended but unusual inversion *++i*. • For example, see FeeController.sol:77 `for (uint256 i; i < paymentTokens.length; i++)`
Mitigation	Replace all instances of *i++* with *++i*. Note that since solidity 0.8.22, this behavior is even more optimized. Additional optimization can be obtained by wrapping the *++i* operator inside an unchecked block.

Incorrect Comment

ID	SAY-Gr-08
Status	Fixed
Risk	Informational
Business Impact	The incorrect comment may be confusing to readers.
Location	- `FeeManager.sol:215; calculateFees(uint256, uint256, uint256, uint256)`
Description	The comment in the function *`calculateFees()`* states `// apply variable fee on baseFee total, set to variableFee` However, *`variableFee`* is not applied to the total of the base fee, but to the whole trading volume. According to the other documentation, this seems to be the intended behavior and it is not intended that the variable fee is a percentage of the base fee.
Mitigation	Change the comment (if the desired behavior is a fee based on the trading volume).

GroupOS Safe

Slot Pointers Can Be Precompiled

ID	SAY-Sa-01
Status	Fixed
Risk	Low
Business Impact	A little bit of gas can be saved by not recalculating the hash each time the function is called.
Location	–

Description	Within *`_hashSafeSensitiveState()`* two slots are calculated by means of keccak256 hash: *`fallbackHandlerSlot`* and *`guardStorageSlot`*.
	`bytes32 fallbackHandlerSlot = keccak256("fallback_manager.handler.address");`
	`bytes32 guardStorageSlot = keccak256("guard_manager.guard.address");`
	
	These are calculated each time *`checkModuleTransaction`* is called.

Mitigation	You can simply calculate the slots ahead of time and put the results in the contract.