Smart Contract Audit Report for Grix

Testers

1. Or Duan
2. Avigdor Sason Cohen

Table of Contents

Management Summary

Grix contacted Sayfer to perform a security audit on their router smart contract in September 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 Grix's router.

Over the research period of 3 weeks, we discovered 20 vulnerabilities in the contract.

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

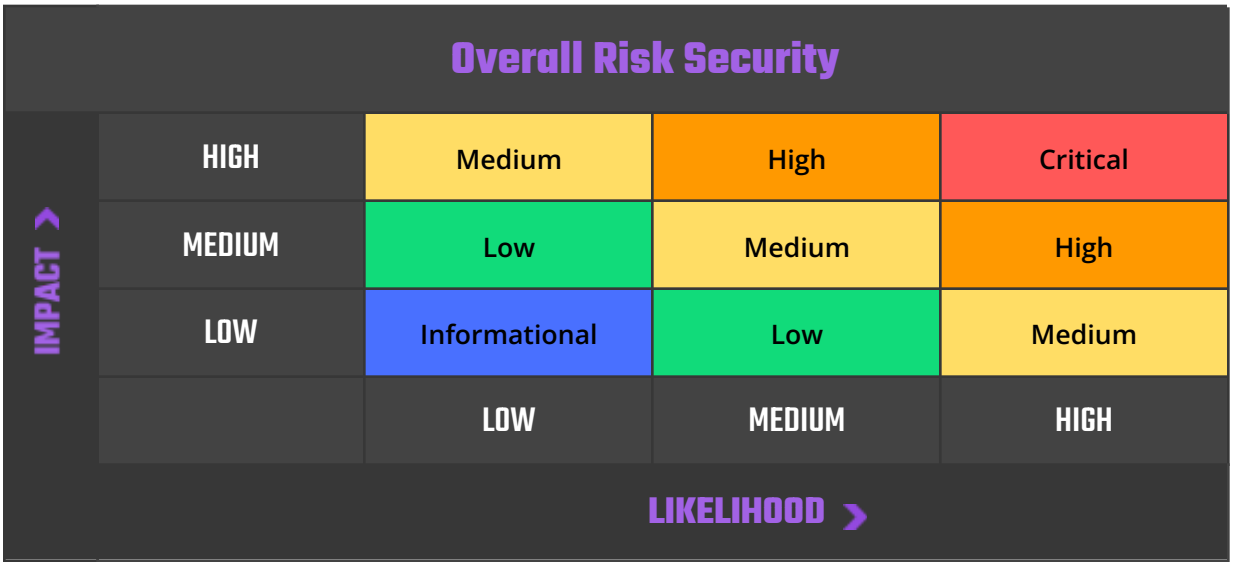
Risk Methodology

At Sayfer, we are committed to delivering the highest quality smart contract audits to our clients. That's why we have implemented a comprehensive risk assessment model to evaluate the severity of our findings and provide our clients with the best possible recommendations for mitigation.

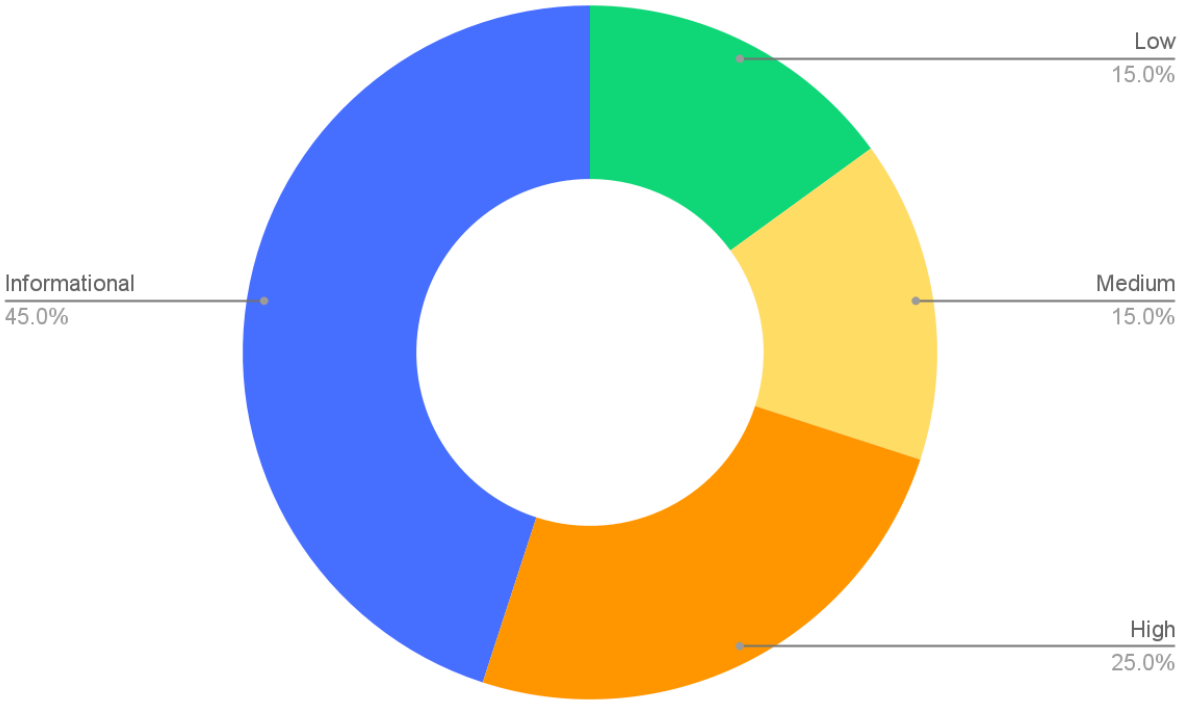
Our risk assessment model is based on two key factors: **IMPACT** and **LIKELIHOOD**. Impact refers to the potential harm that could result from an issue, such as financial loss, reputational damage, or a non-operational system. Likelihood refers to the probability that an issue will occur, taking into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

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

Risk is defined as follows:



Vulnerabilities by Risk



Risk	Low	Medium	High	Critical	Informational
# of issues	3	3	5	0	9

Approach

Introduction

Grix contacted Sayfer to perform a security audit on their router 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 contract.

Scope Overview

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

Contract	SHA-256
`GrixRouter.sol`	`2d85b218ca8d735b8255bdd9fb885f1f576d24c5d89bda115119b045b2a1e634`

Our tests were performed from 02/09/2024 to 23/09/2024.

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.

Protocol Overview

Protocol Introduction

Grix is a unified liquidity protocol for the DeFi options market. Grix aggregates various DeFi options AMMs into a single API & UI, allowing traders to execute trades with deep liquidity and minimal price impact. Optimal trade execution is achieved using a network of solvers that compete to find the best routes for the order flow. Trading is further supported by a dedicated secondary options pool (SOP), which allows makers (secondary options sellers) to sell their options in Grix. The SOP serves as an additional liquidity source for options buyers.

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

`handleRefund(Operation, uint256)` Does Not Support ETH

ID	SAY-01
Status	Acknowledged
Risk	High
Business Impact	Lack of support for ETH (although they are still included in the balance) could lead to reverts when users attempt to refund their funds, locking them in the contract.
Location	- `GrixRouter.sol; handleRefund(Operation, uint256)` - `GrixRouter.sol; getBalance(address, address)`
Description	`handleRefund(Operation, uint256)` substructs the final balance in an operation with the initial balance to calculate the refund amount. To get the final amount, `getBalance(address, address)` is called. ● `GrixRouter.sol:272-278` Interestingly, if a refund is in order, only refunds for ERC20 tokens are supported. ● `GrixRouter.sol:280-291` However, `getBalance(address, address)` supports both ERC20 tokens and ETH. ● `GrixRouter.sol:335-346`

```
uint256 finalBalance = getBalance(
   operation.tokenAddress,
   address(this)
);
refundAmount = finalBalance > initialBalance
   ? finalBalance - initialBalance
   : 0;
```

```
if (refundAmount > 0) {
   IERC20(operation.tokenAddress).transfer(
       operation.userAddress,
       refundAmount
   );
                                  [...]
}
```

```
function getBalance(
```

```
    address asset,
    address account
) internal view returns (uint256) {
    if (asset == address(0)) {
        // ETH balance
        return account.balance;
    } else {
        // ERC20 balance using OpenZeppelin's IERC20
        return IERC20(asset).balanceOf(account);
    }
}
```

Mitigation	Review whether the discrepancy between the two functions is intentional or not. If it isn't, we recommend extending `handleRefund(Operation, uint256)` to support native tokens or alternatively removing ETH support from `getBalance(address, address)`.

Non-Standard EIP-20 Tokens are not supported

ID	SAY-02
Status	Fixed
Risk	High
Business Impact	Refunds involving non-standard tokens such as USDT will lead to reverts.
Location	- `GrixRouter.sol; handleRefund(Operation, uint256)`

Description	`handleRefund(Operation, uint256)` handles refunds for ERC20 tokens by using the standard ERC20 interface:
	• `GrixRouter.sol:280-291`
	`if (refundAmount > 0) {`
	`  IERC20(operation.tokenAddress).transfer(`
	`  operation.userAddress,`
	`  refundAmount`
	`  );`
	`  [...]`
	`}`
	The standard interface expects a boolean to be returned, but some non-standard tokens simply return void, leading to reverts. The most famous example of such a token is probably USDT.

Mitigation	We recommend using OpenZepplin's `safeTransfer(IERC20, address, uint256)` instead.

Signatures Can Be Used to Execute Arbitrary Operations

ID	SAY-03
Status	Fixed
Risk	High
Business Impact	Authorized senders are able to use signatures as blank cheques to execute whichever and however many operations they wish.
Location	- `GrixRouter.sol; executeBatchOperations(Operation[], bytes, Solution[], bytes, uint256)` - `GrixRouter.sol; verifySolutionSign(Solution[], bytes, bytes)`

Description	`executeBatchOperations(Operation[], bytes, Solution[], bytes, uint256)` makes use of `verifySolutionSign(Solution[], bytes, bytes)` to authorize the execution.
	• `GrixRouter.sol:149-157`
	`address signer = verifySolutionSign(`
	`solution,`
	`encodedMessageSolution,`
	`signature`
	`);`
	`require(`
	`isAuthorizedSigner(signer),`
	`"Signature verification failed: The provided signature does not match an authorized signer"`
	`);`
	
	However, this function only takes `solution[]` into account to verify signed messages. That means an authorized sender can use the valid signature and provide any amount of operations to be executed.

Mitigation	It is recommended to include an `operations[]` value in the message signed by the signer, so that the signature can only be used to execute those specific operations.

`verifySolutionSign(Solution[], bytes, bytes)` Lacks Nonce Verification

ID	SAY-04
Status	Open
Risk	High
Business Impact	The lack of a nonce to prevent signature replay enables the abuse of `executeBatchOperations(Operation[], bytes, Solution[], bytes, uint256)`. The same signature can be used over and over again.
Location	- `GrixRouter.sol; verifySolutionSign(Solution[], bytes, bytes)` - `GrixRouter.sol:87-92; struct Solution`
Description	`verifySolutionSign(Solution[], bytes, bytes)` returns the signer address of the message generated from `solution[]`. However, the array does not contain a nonce that prevents from the signature replay. As an effect, the signature can be used multiple times.
Mitigation	It is recommended to add and verify a nonce that is incremented after every valid signature usage to prevent replay attacks.

`initialBalance` Isn't Zeroed on Each Loop Iteration

ID	SAY-05
Status	Fixed
Risk	High
Business Impact	Refund calculation will be broken for non-`checkBalance` operations which follow a `checkBalance` operation in the same batch. Because this has the potential to affect a large number of transactions, we rate this finding as high risk.
Location	- `GrixRouter.sol; executeBatchOperations(Operation[], bytes, Solution[], bytes, uint256)`

Description	When executing a batch of operations, `executeBatchOperations(Operation[], bytes, Solution[], bytes, uint256)` first zeros the initial balance.
	• `GrixRouter.sol:165`
	`uint256 initialBalance = 0;`
	
	The variable is then set to a value from `getBalance(address, address)` for `checkBalance` operations.
	• `GrixRouter.sol:175-180`
	`if (operation.checkBalance) { initialBalance = getBalance( operation.tokenAddress, address(this) ); }`
	
	However, if the loop continues afterward to non-`checkBalance` operations, the now-obsolete value from the earlier iteration remains, and is given to `executeSingleOperation(Operation, uint256)`, which later hands it over to `handleRefund(Operation, uint256)` for its calculation, breaking it for the affected operations.
	
	Namely, because eligibility for a refund and its value is determined by comparison and its subtraction from the final balance, expected refunds may fail to materialize or may be lower than expected.
	• `GrixRouter.sol:276-278`
	`refundAmount = finalBalance > initialBalance ? finalBalance - initialBalance : 0;`

Mitigation	Zero `initialBalance` on every loop iteration.

Signatures without Deadlines

ID	SAY-06
Status	Open
Risk	Medium
Business Impact	A signature could be generated and only used much later, possibly enabling abuse.
Location	- `GrixRouter.sol; verifySolutionSign(Solution[], bytes, bytes)` - `GrixRouter.sol:87-92; struct Solution`
Description	Signatures do not have a usage deadline, so they remain valid indefinitely and could possibly be used in unwanted contexts.
Mitigation	It is recommended to add and verify a short deadline in `verifySolutionSign(Solution[], bytes, bytes)`, e.g. 15 minutes.

Lack of Data Returned Check May Lead to Loss of Funds

ID	SAY-07
Status	Open
Risk	Medium
Business Impact	Contracts that silently fail by returning a boolean may cause accounting pitfalls.
Location	- `GrixRouter.sol; executeSingleOperation(Operation, uint256)`
Description	After performing an operation, `executeSingleOperation(Operation, uint256)` checks whether the low-level call was successful. • `GrixRouter.sol:215-226` `if (operation.callData.length == 0) { // Direct ETH transfer (success,) = operation.targetContractAddress.call{ value: operation.value }(""); } else { // Contract call with callData (success,) = operation.targetContractAddress.call{ value: operation.value }(operation.callData); } require(success, "Operation failed");` But it doesn't check the returned value. Some contracts do not revert on the failure, but return false instead. Thus, a silent failure can be processed by the protocol.
Mitigation	Check the returned value from the transaction as well, not just whether it reverted or not.

No `msg.value` Validation

ID	SAY-08
Status	Open
Risk	Low
Business Impact	Since there is probably no intention to store funds in the contract, trying to withdraw from it will likely lead to reverts. However, the contract can in fact receive funds through the fallback `receive()` function, and in case of insufficient `msg.value` when calling `executeBatchOperations(Operation[], bytes, Solution[], bytes, uint256)`, these funds will be withdrawn to pay for operations, this can obviously be used by malicious actors who wish to tap into these funds.
Location	- `GrixRouter.sol; executeBatchOperations(Operation[], bytes, Solution[], bytes, uint256)`
Description	`executeBatchOperations(Operation[], bytes, Solution[], bytes, uint256)` is marked `payable`, presumably so users can give the contract the money they intend to send in their operations. However, the msg.value is never checked against the cumulative value of the operations, so users can low-ball. In the best case, if the contract holds no funds or insufficient funds to complete the batch, the low-level transfers in `executeSingleOperation(Operation, uint256)` will revert. In the worst case, where enough funds are available in the contract, the contract will pay out of its own pocket.
Mitigation	There are two possible mitigations: - If the contract is never ever supposed to hold funds, then fallback `receive()` function should be reconsidered, but otherwise things can remain as is. - If the contract will sometimes store funds, `executeBatchOperations(Operation[], bytes, Solution[], bytes, uint256)` should make sure that `msg.value` matches the cumulative value of the operations in the batch

Unbounded Loops

ID	SAY-09
Status	Open
Risk	Low
Business Impact	
Location	- `GrixRouter.sol; executeBatchOperations(Operation[], bytes, Solution[], bytes, uint256)` - `GrixRouter.sol; executeSingleOperation(Operation, uint256)`

Description	`executeBatchOperations(Operation[], bytes, Solution[], bytes, uint256)` iterates through the supplied operation array in an unbounded loop.
	• `GrixRouter.sol:167-183`
	`for (uint256 i = 0; i < operations.length; i++) {`
	`[...]`
	`}`
	
	`executeSingleOperation(Operation, uint256)` also performs multiple validations for each operation by iteration in an unbounded loop.
	• `GrixRouter.sol:232-245`
	`for (uint256 i = 0; i < operation.validations.length; i++) {`
	`[...]`
	`}`
	
	In both cases, if the number of operations and validations, respectively, is too large, the function may run out of gas and revert.

Mitigation	Consider setting reasonable upper limits for the number of operations and validations.

Non-Compliance with EIP-712

ID	SAY-10
Status	Open
Risk	Low
Business Impact	Protocols are always encouraged to follow standard practices wherever possible. Because they are well-tested and peer-reviewed, they are on the whole safer and more efficient than homegrown solutions.
Location	- `GrixRouter.sol; verifySolutionSign(Solution[], bytes, bytes)`
Description	EIP-712 is a procedure for hashing and signing of typed structured data as opposed to just bytestrings. One of the mechanisms employed by the EIP is a domain separator, which prevents signatures collisions, where a signature meant for one dApp can work in another. As things stand, the protocol does not contain a domain separator for its Solution struct, and is therefore non-compliant.
Mitigation	We recommend aligning your implementation with the EIP-712 standard. A good reference is OpenZepplin's implementation, which could be either inherited by or used as a guide for your implementation.

Redundant Check

ID	SAY-11
Status	Open
Risk	Informational
Business Impact	The inclusion of the extra check and its associated parameter can unnecessarily increase gas consumption.
Location	- `GrixRouter.sol; verifySolutionSign(Solution[], bytes, bytes)`

Description	`verifySolutionSign(Solution[], bytes, bytes)` checks whether the provided `encodedMessageSolution` parameter matches with value calculated from the solution.
	• `GrixRouter.sol:316-320`
	`require( keccak256(encodedMessageSolution) == keccak256(encodedMessageSolutionWithChain), "Solver payload mismatch: serialized solution does not match the expected payload" );`
	
	But the hash is later verified against the signature by `ECDSA.recover()`, rendering the first check unnecessary.
	• `GrixRouter.sol:323-325`
	`bytes32 hash = keccak256(encodedMessageSolution) .toEthSignedMessageHash(); address recoveredAddress = hash.recover(signature);`

Mitigation	The first manual check and its associated parameter, `encodedMessageSolution`, may be safely removed.

Duplicate Assertions

ID	SAY-12
Status	Open
Risk	Informational
Business Impact	Duplicated code can unnecessarily increase gas consumption.
Location	- `GrixRouter.sol; executeBatchOperations(Operation[], bytes, Solution[], bytes, uint256)` - `GrixRouter.sol; executeSingleOperation(Operation, uint256)`
Description	The assertion `operation.targetContractAddress ≠ address(0)` is done twice for each operation: - First in `executeBatchOperations(Operation[], bytes, Solution[], bytes, uint256)`, lines 169-172 - Then in `executeSingleOperation(Operation, uint256)`, lines 210-213
Mitigation	One of the duplicated assertions can be removed.

Floating Pragma

ID	SAY-13
Status	Open
Risk	Informational
Business Impact	According to industry best practices, floating pragmas should be avoided if possible. Changes in the compiler version could lead to unexpected behavior in the code, so it's best to restrict a contract to the environment it was developed and tested in.
Location	- `Project-wide`
Description	The project has floating pragmas (^0.8.0).
Mitigation	Choose a specific solidity version (the current is obviously recommended) and enforce it. Upgrades to the required compiler version should only be done after reviewing and retesting the code.

Unused Variable

ID	SAY-14
Status	Open
Risk	Informational
Business Impact	Unused variables can lead to unnecessarily increased gas consumption.
Location	- `GrixRouter.sol; handleRefund(Operation, uint256)` - `GrixRouter.sol; executeSingleOperation(Operation, uint256)`
Description	The variable `refundAmount` in `executeSingleOperation(Operation, uint256)` is set to the return value of `handleRefund(Operation, uint256)`, but is never used.
Mitigation	One possibility is that `refundAmount` was supposed to be used to emit the `OperationExecuted` event, but this was overlooked. If this is not the case, `refundAmount` can be removed and `handleRefund(Operation, uint256)` does not have to return anything.

No Support for Chain Forks

ID	SAY-15
Status	Open
Risk	Informational
Business Impact	In case of a chain fork, `verifySolutionSign(Solution[], bytes, bytes)` will no longer function correctly. Due to the unlikeliness of the event, however, this finding has been assessed to be informational.
Location	- `GrixRouter.sol:28` - `GrixRouter.sol; verifySolutionSign(Solution[], bytes, bytes)`
Description	The `chain` variable is set in the constructor and is later used in `verifySolutionSign(Solution[], bytes, bytes)`. In the unlikely case of a chain fork, the value of this variable will no longer be correct, and `verifySolutionSign(Solution[], bytes, bytes)` may malfunction.
Mitigation	Add a setter for the `chain` variable.

Access Control Recommendations

One-Step Ownership Transfer

ID	SAY-ACC-01
Status	Open
Risk	Low
Business Impact	Passing in a wrong address when transferring the ownership will result in the irrevocable loss of the ownership role.
Location	- `GrixRouter.sol`
Description	`transferOwnership(address)` provided by OpenZepplin's Ownable immediately sets the new owner to the passed address, without requiring a confirmation from this account / contract. If a wrong address is passed, the Owner role will be instantly and irrecoverably lost.
Mitigation	To avoid errors when performing an ownership transfer, it is recommended to implement a two-step transfer. The current owner should only be able to set a new pending owner. This address then needs to accept the ownership by calling a dedicated function. An off the shelf solution, `Ownable2Step`, is also offered by OpenZepplin.

Both `AccessControl` and `Ownable` are Used

ID	SAY-ACC-02
Status	Open
Risk	Informational
Business Impact	Unnecessary gas-usage and code complexity.
Location	- `GrixRouter.sol`
Description	`GrixRouter` inherits from both `AccessControl` and `Ownable`. The `DEFAULT_ADMIN_ROLE` from `AccessControl`, which has the ability to change all other roles, can possibly serve as a suitable replacement. Alternatively, an owner role could be defined.
Mitigation	Remove `Ownable` and use `AccessControl` to replace it. If this is done, SAY-ACC-01 can be ignored.

Implementing `AccessControlDefaultAdminRules` Should Be Considered

ID	SAY-ACC-03
Status	Open
Risk	Informational
Business Impact	Implementing AccessControlDefaultAdminRules can provide extra security assurances.
Location	- `GrixRouter.sol`
Description	`AccessControlDefaultAdminRules` is a useful extension for OpenZepplin's `AccessControl` that implements a few extra security measures for the sensitive `DEFAULT_ADMIN_ROLE`: - Only one account holds the `DEFAULT_ADMIN_ROLE` from deployment until it's potentially renounced. - A 2-step process is required to transfer the `DEFAULT_ADMIN_ROLE` to another account. - Enforces a configurable delay between the two steps, with the ability to cancel before the transfer is accepted. - The delay can be changed by scheduling - It is not possible to use another role to manage the `DEFAULT_ADMIN_ROLE`.
Mitigation	Consider replacing `AccessControl` and `Ownable` with `AccessControlDefaultAdminRules`. SAY-ACC-01 and SAY-ACC-02 could then be safely ignored. Refer here for more information.

Authorized Sender Roles Cannot be Removed

ID	SAY-ACC-04
Status	Open
Risk	Informational
Business Impact	As explained in SAY-ACC-05, AccessControl already has `revokeRole` functions by default, so other than being inconsistent, this finding has no further implications.
Location	- `GrixRouter.sol`
Description	`GrixRouter` implements a `setAuthorizedSender(address)` but lacks a similar function for removing the same role. This may be an oversight.
Mitigation	Verify whether authorized sender roles should be irrevocable.

Unnecessary Role Getters/Setters

ID	SAY-ACC-05
Status	Open
Risk	Informational
Business Impact	The functions, aside from being unnecessary, clutter up the code and lead to increased gas consumption.
Location	- `GrixRouter.sol; isAuthorizedSigner(address)` - `GrixRouter.sol; setAuthorizedSender(address)` - `GrixRouter.sol; addAuthorizedSigner(address)` - `GrixRouter.sol; removeAuthorizedSigner(address)`
Description	The aforementioned functions simply call `AccessControl`'s default functions and can simply be removed.
Mitigation	Consider removing the listed functions.

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