SAYFER

Smart Contract Audit Report for Dusa

Testers

1. Or Duan
2. Avigdor Sason Cohen

Table of Contents

Management Summary

Dusa 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 Dusa’s smart contracts.

Over the research period of 4 weeks, we discovered 17 vulnerabilities in the contract.

Overall, Dusa is a well-built protocol. The fact that it is derived and translated from TraderJoe makes it a very solid protocol with a fairly common architecture, but considered optimal. However, we do have a few recommendations which we feel could improve the quality and security of the protocol. These recommendations are detailed in the "Architecture Review" section.

In conclusion, several fixes should be implemented following the report, but 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 or acknowledged by the Dusa team.

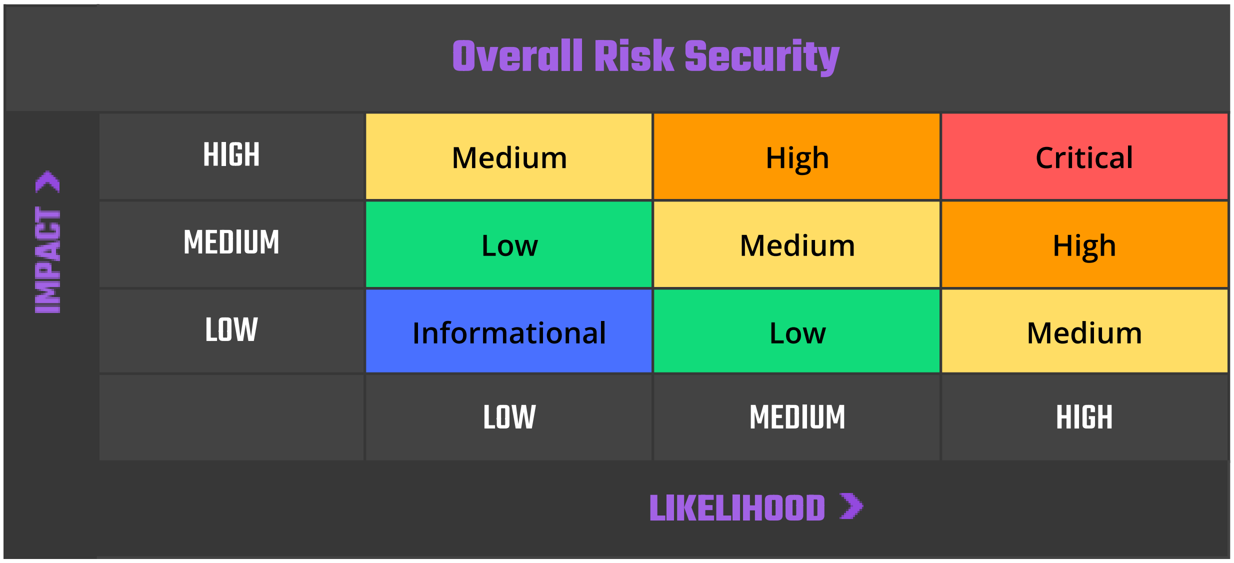
Risk Methodology

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.

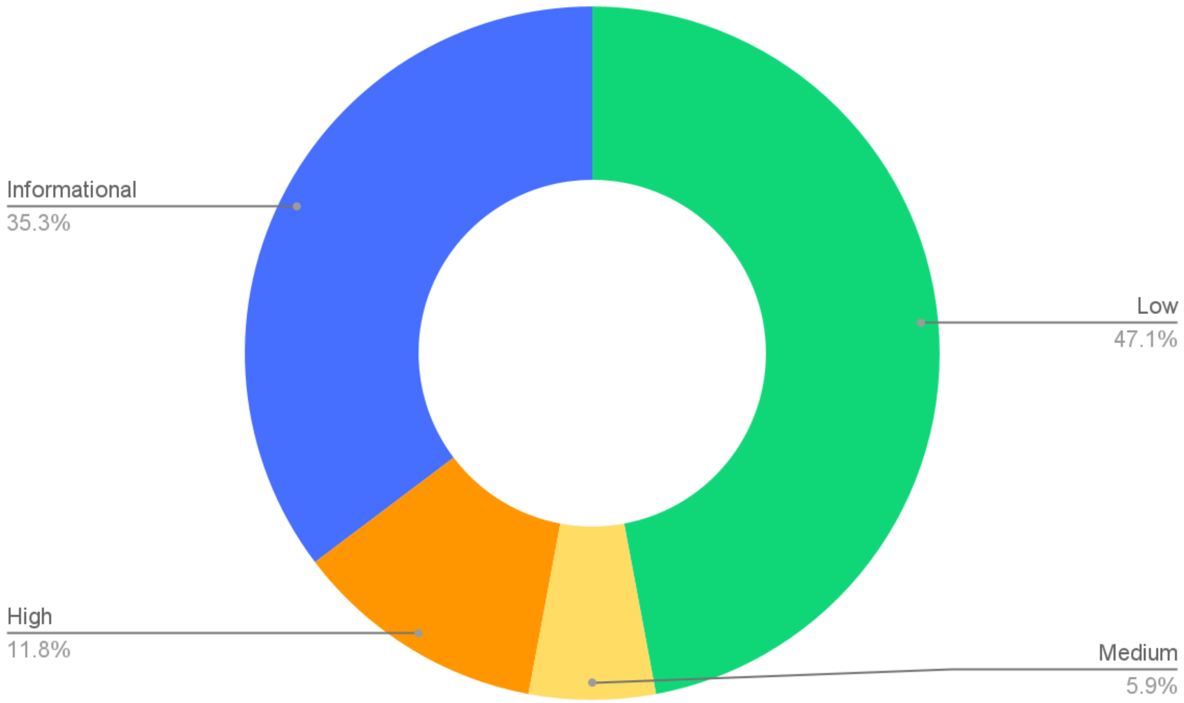
Risk is defined as follows:



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	8	1	2	0	6

Approach

Introduction

Dusa 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.

Contract	Sha-256
`Factory.ts`	`b86a4c350c37a0d8c62495bde0be2b0dca2c8a9f343149238548690df5104cee`
`main.ts`	`d0073078a6ef26a402bcfb368e94dfb9dddc3f2a629eabc842e2b62d62772b6a`
`Pair.ts`	`186f3bd5fab92cb4e98ea5daef31f44d1889f28e683663d056ff6649685ce36c`
`Quoter.ts`	`b51c762d8ebe50c6b966c9b52a33819a674fb7cf056062ebe34d8f425d879aad`
`Router.ts`	`038e32c21432ad173cf8f3b77a586f222a99a8d0f8e033e40145643671ff87ee`
`WMAS.ts`	`bcef456ec5e27527ae560e21b00fb9f4c8300016aae584da3aa1e869c7018832`

Our tests were performed between January to February 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

Dusa is a groundbreaking decentralized finance (DeFi) protocol, introducing a fully decentralized and advanced automated market maker (AMM) experience. Built on the Massa blockchain, Dusa emphasizes key principles such as full decentralization, accessibility for all user profiles, interoperability with future decentralized applications (DApps), and a strong commitment to security and censorship resistance. Its roadmap outlines a phased approach, including research and development, testing, incentivized quests, and the deployment of a decentralized exchange (DEX). Dusa's unique features include concentrated liquidity, variable fees, autonomous liquidity, and complex trading orders, all designed to enhance user experience and yield maximization. With a dedicated team, strategic investors, and advisors, Dusa aims to pioneer the adoption of autonomous smart contracts and extend its decentralized solutions to other ecosystems.

Architecture Review

Acknowledging that Dusa is a derivative of a well-established solution, TraderJoe, and aligns with a recognized architectural pattern, we note limited additional recommendations for the current architecture. However, our evaluation of the Dusa protocol architecture has led to a comprehensive view, combining valuable insights with thoughtful recommendations for improvement:

- From a best practices standpoint, we recommend considering the implementation of a multisig wallet for managing privileged roles in the future. While we acknowledge the recent release of the Massa multisig code (https://github.com/massalabs/massa-standards/tree/feature/multisig-sc/smart-contracts/assembly/contracts/multisig), we advise prudence and consider the use of a standard private key also as a viable option for the present.

- Another insightful recommendation suggests adopting a unified liquidity model, envisioning a global Vault/PoolManager instead of individual contracts per pool. This strategic shift, inspired by recent advancements in AMMs like Uniswap V4 and Balancer, promises significant advantages. These include cost-effective multi-hop swaps and enhanced flashloan capacity, attributed to streamlined access to the entire liquidity pool.

Security Evaluation

Dusa-Specific Tests

As Dusa is in a unique situation of being a reflection of TraderJoe in a different programming language, we performed multiple additional tests in addition to our usual ones to ensure there are no missing parts. Here is a list of tests performed specifically for the Dusa protocol:

Security Checks

- Overflow/underflow checks, especially due to the change in codebase from a language with default checks to one where explicit checks are required.
- Usage of safe libraries (checked_*).
- Consistency between contracts regarding the flow of funds and calls.
- Visibility of functions to ensure that the proper functions are exported.
- Access control implementation and protection of sensitive functions.
- Swap protection to secure interactions with liquidity from slippage.

Authentication and Authorization

- Audit of the entire authentication process, owner rights, and individual public entry-points authorization schema.
- Identification of potential vulnerabilities leading to malicious permissions for attackers.

Math and Variable Handling

- Calculation-related issues, especially with multiple types of variables, casting, and managing floating-point numbers.
- Decimal errors related to the unique aspect of MASSA having 9 decimals.
- Handling of overflow/underflow capabilities.

Flash Loan and Liquidity Management

- Examination of typical mistakes related to flash loans, including miscalculations and fund blocking.
- Issues related to factory and pair creation, such as creating dangerous pools and liquidity management.

- Verification of liquidity removal calculations, considering fees obtained during liquidity provision.

Functionality and Implementation

- Implementation of swaps compared to other DEXs, focusing on fee management and differences in token types.
- Possibility of DoS attacks when operating on unsafe types like Vectors or Arrays.
- Verification of centralization issues, such as potential influence on prices, user token transfers, and configuration changes.

Context Data Handling

- Verifying that the correct context data is passed to functions to avoid vulnerabilities.

Generic Tests

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	**Test Name**

Service	
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.

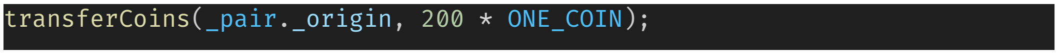
Test Coverage	**Test Name**
G12.2	Abuse narratives detailed in the threat model are covered by unit tests.
G12.3	Sensitive functions in verified contracts are covered with tests in the development phase.
G12.4	Implementation of verified contracts has been checked for security vulnerabilities using both 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

[H] Front Running Risk when Creating a New Pair

ID	SAY-01
Status	Fixed
Risk	High
Business Impact	In some cases, users may lose the funds they deposited in order to fund pair creation. Either a malicious frontrunner may create a pair at the expense of another user, or multiple users could attempt to create a pair at the same time, succeeding or failing randomly.
Location	- `Router.ts; createLBPair(StaticArray<u8>)` - `Factory.ts; createLBPair(StaticArray<u8>)`
Description	In order to create a new pair, the user has to call `Router::createLBPair(StaticArray<u8>)`. Under the hood, this function calls `Factory::createLBPair(StaticArray<u8>)`, which in turn calls Massa's `transferCoins(Address, number)`. ● `Factory.ts:264; createLBPair(StaticArray<u8>)`  `transferCoins(_pair._origin, 200 * ONE_COIN);` According to the documentation, this function simply transfers coins from the current address to a given address. That means that these funds have to be previously sent to `Factory`. On the other hand, users have the ability to call createPair from Router, which is the default entry point for other operations. However, if they are deposited in any separate transaction, there is a possibility that someone, intentionally or otherwise, will also call `createLBPair(StaticArray<u8>)`. When a block with transactions is finalized, they will be laid out in an unknown order. Then, the first transaction included, will have these funds transferred from Factory to a new pair, while the second user (the latter transaction) will have their transaction reverted due to lack of available funds.

Mitigation	The funding of pair creation has to actually take place in the same transaction to avoid transaction reordering issues. For example, one of the ways to achieve it could be a mechanism checking the factory balance & the pool balance at the end of a function and sending back the surplus, similar to this https://github.com/massalabs/coin-vester/blob/master/smart-contract/assembly/contracts/main.ts#L34 The mechanism will be in place in the `createLBPair` function (of `Factory`), in the `constructor`, `mint`, `burn`, `collectFees`, `increaseOracleLength`, `safeTransferFrom` & `safeBatchTransferFrom` (of pair). These are the functions able to spend Massa for storage as they can create new entries in the storage (modifying storage doesn't cost anything). Functions interacting with these functions also need to have this mechanism. They can be separated into two different kinds: - Functions interacting with these and which do not interact with WMAS (`createLBPair` from router & factory + `addLiquidity`, `removeLiquidity`) will transfer all the `transferredCoins`, save the balance of the SC in a constant, and after the SC call, transfer back to the caller the difference between the new balance and the saved balance. - Functions interacting with these and which interact with WMAS (`addLiquidityMAS`, `removeLiquidityMAS`) will need one more parameter to know the number of coins that need to be wrapped, you will send the difference between `transferredCoins` and this number in the call. The rest of the mechanism in these functions will be the same as the one just before.

[H] Funds Deposited for Pair Creation May Be Lost

ID	SAY-02
Status	Fixed
Risk	High
Business Impact	Users who deposit funds to the router to create a pair may lose their money without seeing a result. We rated this issue as high (rather than critical) because an alternate, working flow exists – calling `factory::createLBPair(StaticArray<u8>)` directly with the funds.
Location	- `Router.ts; createLBPair(StaticArray<u8>)` - `Factory.ts:264; createLBPair(StaticArray<u8>)`
Description	This issue is a result of how pair creation is designed. A user can call `Router::createLBPair(StaticArray<u8>)` and since a Pair creation requires a deposit, he can attach funds to the function call, which sounds like an intuitive solution. However, if the user does so, the funds will never reach the pair and will be lost in the contract (though can be rescued by the owner). This is because they will be stuck in the Router contract, while the pair creation takes place in the Factory contract, where transferCoins(), which deducts funds off the current (the factory's) address, is used instead. ● `Factory.ts:264; createLBPair(StaticArray<u8>)` `transferCoins(_pair._origin, 200 * ONE_COIN);` Since it's implemented in the factory, funds sent to the router won't reach the factory, which also won't have the funds to create a pair and revert.
Mitigation	One option is to implement the solution suggested in `SAY-01`. Conversely, `Factory.ts` can be called directly to create pairs and require users to send either the exact required amount along with the function call, OR at least the required amount with refunds of any surplus funds sent.

[M] Multiple Pairs of the Same Type May Dilute Liquidity

ID	SAY-03
Status	Fixed
Risk	Medium
Business Impact	If an AMM application allows for creating multiple pools of the same parameters, the liquidity may be diluted across these pools, instead of being concentrated in just one. This might negatively impact users and might result in less favorable trades available for protocol users. In addition, the previous pair won't be used for routing anymore. This can result in liquidity or price manipulation.
Location	- `Factory.ts; createLBPair(StaticArray<u8>)` - `Router.ts; createLBPair(StaticArray<u8>)`
Description	When creating a new Pair, there is no check if exactly the same pair already exists. This may scatter users' liquidity across multiple equivalent pools. Low liquidity pools are more prone to price manipulations and offer less favorable trades, and should therefore be avoided.
Mitigation	Check if a pair of certain parameters already exist on pair creation, if so, revert.

[L] Exact Balance Check after a Flash Loan May Be Abused

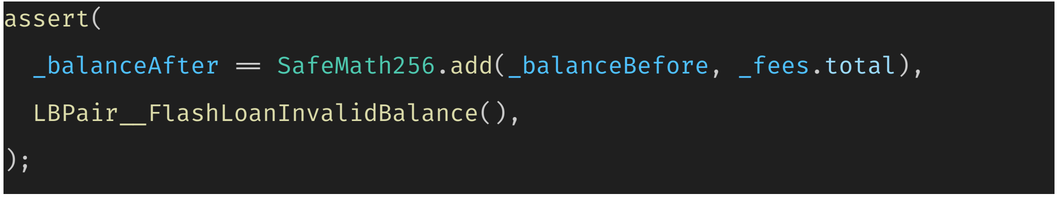
<table>
<tr><td>ID</td><td>SAY-04</td></tr>
<tr><td>Status</td><td>Fixed</td></tr>
<tr><td>Risk</td><td>Low</td></tr>
<tr><td>Business Impact</td><td>A flash loan receiver might perform external calls in the callback. Because the system requires the balance after the flash loan to be exactly the balance before plus the fees, these external contracts could transfer a very small amount to the pair during the execution to ensure that the flash loan will fail (which may be profitable for them in certain situations).</td></tr>
<tr><td>Location</td><td>

- `Pair.ts:280; flashLoan(StaticArray<u8>)`

</td></tr>
<tr><td>Description</td><td>

After a flash loan, the system requires that the balance is exactly the balance before the flash loan plus fees.

- `flashLoan(StaticArray<u8>)`



```
assert(
  _balanceAfter == SafeMath256.add(_balanceBefore, _fees.total),
  LBPair__FlashLoanInvalidBalance(),
);
```

Typically, it is required that the balance is greater, as requiring an exact match is restrictive.

</td></tr>
<tr><td>Mitigation</td><td>Only check if the balance is greater than the balance before the flash loan plus fees. This is also the logic that TraderJoe uses within their flash loan function.</td></tr>
</table>

[L] Ignoring an LB Pair does not Emit an Event

ID	SAY-05
Status	Fixed
Risk	Low
Business Impact	Off-chain indexers may receive incorrect information, leading to wrong calculations.
Location	- `Factory.ts; setLBPairInformation(StaticArray<u8>)`
Description	The function `setLBPairIgnored(StaticArray<u8>)` can be used to control whether a pair should be ignored for routing or not. Any change in the flag via this function emits the event `SET_LBPAIR_IGNORED`. However, it is also possible to change the flag via setLBPairInformation(StaticArray<u8>), which does not emit an event.
Mitigation	Always emit an event when changing the flag to ensure that off-chain systems know which pairs will be considered for routing.

[L] Inconsistent Fee Collection

<table>
<tr><td>ID</td><td>SAY-06</td></tr>
<tr><td>Status</td><td>Acknowledged</td></tr>
<tr><td>Risk</td><td>Low</td></tr>
<tr><td>Business Impact</td><td>Any user can trigger the fee collection for other users, whereas the protocol fee collection can only be triggered by the owner.</td></tr>
<tr><td>Location</td><td>- Pair.ts; collectFees(StaticArray<u8>)
- Pair.ts:723-726; collectProtocolFees(StaticArray<u8>)</td></tr>
<tr><td>Description</td><td>Anyone can call collectFees(StaticArray<u8>) for any address to distribute the fees to this address. This may be undesirable for certain users who want to control when the fees are distributed to them.

This is additionally inconsistent with collectProtocolFees(StaticArray<u8>), which can only be called by the recipient.

● collectProtocolFees(StaticArray<u8>)
<pre>
assert(
  Context.caller().equals(_feeRecipient),
  LBPair__OnlyFeeRecipient(_feeRecipient, Context.caller()),
);
</pre></td></tr>
<tr><td>Mitigation</td><td>Consider changing the logic in collectFees(StaticArray<u8>) to match collectProtocolFees(StaticArray<u8>), i.e. only allowing the recipient to collect their fees.</td></tr>
</table>

[L] Inconsistent Usage of `SafeMath`

ID	SAY-07
Status	Fixed
Risk	Low
Business Impact	Overflows or underflows in the code may lead to unexpected reverts and/or miscalculations.
Location	—
Description	Even though `SafeMath` is used in many places, certain functions that use arithmetic operations are left unprotected, which may consequently lead to overflows or underflows.
Mitigation	Although no place has been identified where such a situation could be likely, due to the large size of the codebase and the multitude of calculations, we recommend implementing `SafeMath` in as many operations as possible.

[L] Inefficient Asset Enumeration

<table>
<tr><td>ID</td><td>SAY-08</td></tr>
<tr><td>Status</td><td>Fixed</td></tr>
<tr><td>Risk</td><td>Low</td></tr>
<tr><td>Business Impact</td><td>Some factory functions will use up more gas than necessary. In the worst case, if the owner registers a very large amount of assets, this could lead to a DoS. But this is unlikely.</td></tr>
<tr><td>Location</td><td>

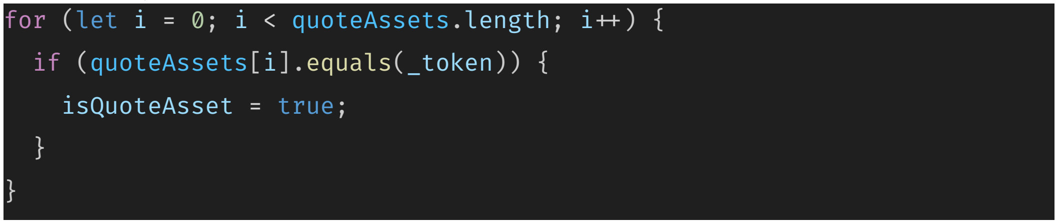
- `Factory.ts:102-106; isQuoteAsset(Address)`

</td></tr>
<tr><td>Description</td><td>

There are two issues with this function:

1. The loop doesn’t break after finding an asset and needlessly iterates over the rest of the array. Thus, every case **is** the worst case.
2. A mapping would be more efficient here, as it has a constant time-lookup.

- `isQuoteAsset(Address)`



```
for (let i = 0; i < quoteAssets.length; i++) {
  if (quoteAssets[i].equals(_token)) {
    isQuoteAsset = true;
  }
}
```

</td></tr>
<tr><td>Mitigation</td><td>

There are two solutions here:

1. Keep the current linear search algorithm and simply add a break in the `if` block after setting `isQuoteAsset` to true.
2. A better solution would be to switch over to a mapping. When a new token is added into the system, `quoteAssets[token]` should be set to true. Henceforth, it can be accessed when necessary – no need to search for it. Perhaps the whole function can simply be refactored out.

</td></tr>
</table>

[L] Maximum Flash Loan Fee is not Set after Deployment

ID	SAY-09
Status	Fixed
Risk	Low
Business Impact	It is possible to have a flash loan fee that is higher than `MAX_FEE`.
Location	- `Factory.ts:80; constructor(StaticArray<u8>)`
Description	Whenever the flash loan fee is set using setFlashLoanFee(), it is validated that it is smaller than the value MAX_FEE. However, this check is not performed when the contract is initialized and it is possible to set an arbitrarily high flash loan fee there.
Mitigation	Validate the flash loan fee in the constructor.

[L] Incomplete Slippage Protection – Missing Deadline Check

ID	SAY-10
Status	Fixed
Risk	Low
Business Impact	There is an increased likelihood that users will get less favorable trades when using the AMM.
Location	`Router.ts` - `addLiquidity(bs: StaticArray<u8>)` - `addLiquidityMAS(bs: StaticArray<u8>)`
Description	Slippage is a result of a volatility in AMM-type solutions, and simply means that the price when requesting a trade/swap/liquidity change is different than when the transaction is executed. To protect users from enormous slippage, which can be natural or a result of intentional attacks like Sandwiching, there are two generic types of protections: minimal amount, which is present, and deadline check, which is missing. Those checks should be implemented on any swap, liquidity addition, and removal, as all of them are subject to slippage. A deadline check simply does not allow the transaction to be included after a certain time point, which for example can intentionally be included later by a malicious block builder to still try to execute the transaction in less favorable price conditions. However, since a minimal token amount is checked, then the impact of this issue is limited.
Mitigation	Implement `_ensure(deadline)` in the same way it is already present in all other functions dealing with liquidity.

[L] One-Step Ownership Transferral

ID	SAY-11
Status	Fixed
Risk	Low
Business Impact	Providing an incorrect address could lead to irrevocable loss of contract ownership.
Location	- `Factory.ts; transferOwnership([u8])`

Description

In `Factory.ts`, change of ownership is achieved by the old owner calling the `transferOwnership([u8])` with the new owner. The function then changes the `OWNER` state variable.

- `transferOwnership([u8])`

```
export function transferOwnership(bs: StaticArray<u8>): void {
  _onlyOwner();

  const _newOwner = new Address(new Args(bs).nextString().unwrap());
  _setFeeRecipient(_newOwner);
  Storage.set(OWNER, _newOwner.toString());
}
```

Mitigation

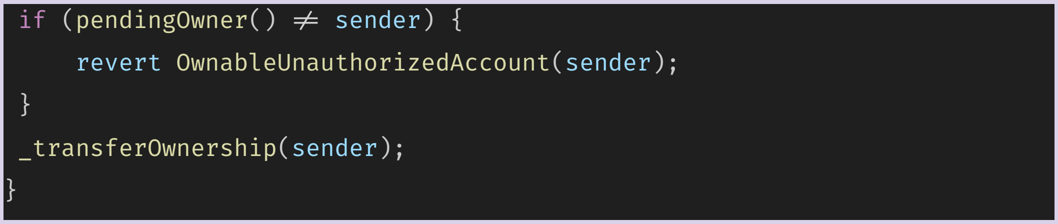
We recommend implementing a two-step process for ownership transfer. This is in line with widely recognized good practices and could prevent accidental transfers. To illustrate, we will use OpenZepplin's implementation.

A two step ownership transfer first requires the current owner to initiate the transfer.

```
function transferOwnership(address newOwner) public virtual override
onlyOwner {
 _pendingOwner = newOwner;
 emit OwnershipTransferStarted(owner(), newOwner);
}
```

But in order to complete the process, the new owner has to approve the transfer.

```
function acceptOwnership() public virtual {
 address sender = _msgSender();
```



```
 if (pendingOwner() ≠ sender) {
     revert OwnableUnauthorizedAccount(sender);
 }
 _transferOwnership(sender);
}
```

This ensures that the new owner is the intended target.

[I] Insufficient Event Emission

ID	SAY-12
Status	Fixed
Risk	Informational
Business Impact	Decreased traceability and visibility of historical key state changes affecting the protocol.
Location	`Factory.ts` - `setLBPairInformation(bs: StaticArray<u8>)` - `forceDecay(bs: StaticArray<u8>)` - `transferOwnership(bs: StaticArray<u8>)`
Description	Some of the key state changes of the protocol do not emit an associated event. **Note:** `transferOwnership(bs: StaticArray<u8>)` does emit an event, but only for `_setFeeRecipient`.
Mitigation	Add event emission in the specified locations.

[I] Getters and Setters are Mixed up with External Functions

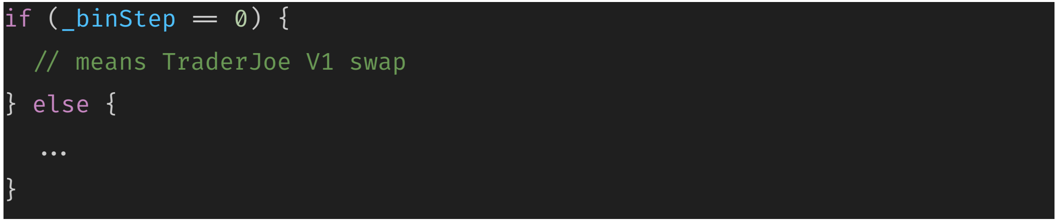
ID	SAY-13
Status	Fixed
Risk	Informational
Business Impact	Mixing, getters, setters, entrypoints and internal functions make the code less readable.
Location	- `Factory.ts:631, 639` - `Pair.ts:1031, 1227`
Description	Certain contracts have separate sections containing functions that change the state (setters), read it (getters), or are part of the function call tree (internal). In some contracts, however, this order is disturbed - the functions are mixed up. Sometimes, exported functions (entrypoints) are also mixed up with internal ones.
Mitigation	We recommend that sections in the code be properly separated, entry points separated from internal functions, and that the convention of using the _ sign before private functions be maintained.

[I] Lack of Unit Tests

ID	SAY-14
Status	Acknowledged
Risk	Informational
Business Impact	Test scenarios and unit tests help developers detect bugs and vulnerabilities that would otherwise slip by in a simple static analysis. Dynamic testing is invaluable in ensuring the quality of the finished product.
Location	—
Description	Apart from a dozen or so tests for a number of library functionalities, the code did not have unit, functional or integration tests. Not all flaws can be detected by simple static analysis, leaving the product in a precarious position.
Mitigation	It is considered good practice to ensure that code coverage for tests is as high as possible and that test scenarios include both happy and unhappy paths.

[I] Unnecessary Code

ID	SAY-15
Status	Fixed
Risk	Informational
Business Impact	This issue has no direct impact on the code or on the protocol, and was therefore rated informational.
Location	- `Router.ts:770-772; _getAmountsIn(u64[], Address[], IERC20[], u256)`

Description	The function _getAmountsIn() takes the bin steps as an argument and performs the following check:
	● `_getAmountsIn(u64[], Address[], IERC20[], u256)`
	
	`if (_binStep == 0) {`
	`  // means TraderJoe V1 swap`
	`} else {`
	`  ...`
	`}`
	However, this is unnecessary. The system does not support V1-style swaps and the rest of the function does not consider the bin steps, as it receives the pairs (corresponding to the bin steps) as an additional argument.

Mitigation	Consider removing the argument _pairBinSteps and the unnecessary if-clause.

[I] Unused Imports

ID	SAY-16
Status	Fixed
Risk	Informational
Business Impact	This issue has no direct impact on the code or on the protocol, and was therefore rated informational.
Location	- `Pair.ts:35,` - `Router.ts:4,` - `WMAS.ts:5,6,` - `interfaces/IPair.ts:5`
Description	The specified imports appear to never be used in their respective contracts. They could be removed.
Mitigation	Remove the unused imports.

[I] Usage of ‘Magic Numbers’

ID	SAY-17
Status	Acknowledged
Risk	Informational
Business Impact	This may increase the difficulty of parsing the code for unfamiliar readers.
Location	- `Pair.ts:113` - `Factory.ts:264, 735`
Description	In several places, the contract uses numerical values that are not well documented, described or commented on.
Mitigation	Add comments explaining the choice of constants.