CERTIK

CertiK Audit Report for Amun AG

Contents

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and Amun AG (the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement"). This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, has developed a proprietary Formal Verification technology to apply rigorous and complete mathematical reasoning against code. This process ensures algorithms, protocols, and business functionalities are secured and working as intended across all platforms.

CertiK differs from traditional testing approaches by employing Formal Verification to mathematically prove blockchain ecosystem and smart contracts are hacker-resistant and bug-free. CertiK uses this industry-leading technology together with standardized test suites,

static analysis, and expert manual review to create a full-stack solution for our partners across the blockchain world to secure 6.2B in assets. For more information: https://certik.org.

Executive Summary

This report has been prepared for **Amun AG** to discover issues and vulnerabilities in the source code of their **Jasper Platform Smart Contracts**. A comprehensive examination has been performed, utilizing Dynamic Analysis, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Testing Summary

SECURITY LEVEL

Smart Contract Audit

This report has been prepared as a product of the Smart Contract Audit request by Amun AG.

This audit was conducted to discover issues and vulnerabilities in the source code of Amun AG's Jasper Platform Smart Contracts.

TYPE	Smart Contracts & Token
SOURCE CODE	https://github.com/amun/jasper-contracts
PLATFORM	EVM
LANGUAGE	Solidity
REQUEST DATE	March 26, 2020
REVISION DATE	April 10, 2020
METHODS	A comprehensive examination has been performed using Dynamic Analysis, Static Analysis, and Manual Review.

Review Notes

Introduction

CertiK team was invited by the Amun AG team to audit the design and implementations of its Jasper Platform smart contracts. The audited source code link is:

- https://github.com/amun/jasper-contracts/tree/0625f0fdbacdcaaaa60634774310494c98cfe3b3

The goal of this audit was to review the Solidity implementation for its business model, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

The findings of the initial audit were conveyed to the team behind the Jasper implementation and the source code was re-evaluated after the team tackled the issues pinpointed in the initial report. The re-evaluated source code link is:

- https://github.com/amun/jasper-contracts/tree/2e9a57a771fa33874135a527fd396ac21b148491

A final re-evaluation occurred to ensure that the 3 newly found notes were addressed at the following source code link:

- https://github.com/amun/jasper-contracts/tree/b7c2cb890a8510c9b1764fdfcf5d7f6f326cb471

Documentation

We used the following sources of truth about how the Jasper Platform should work:

1. DApp Architecture:
https://github.com/amun/jasper-contracts/blob/0625f0fdbacdcaaaa60634774310494c98cfe3b3/README.md#general-dapp-arichitecture
2. Smart Contract Technical Overview:
https://github.com/amun/jasper-contracts/blob/0625f0fdbacdcaaaa60634774310494c98cfe3b3/contracts/README.md#smart-contract-overview

These were considered the specification, and when discrepancies arose with the actual code behaviour, we consulted with the Amun AG team or reported an issue.

Summary

The codebase of the project attempts to fulfill a use case that is intricate and ambitious and as such, **inefficiencies and flaws in both the design and implementation** of the various components were identified and properly documented.

While **most of the issues pinpointed were of negligible importance** and mostly referred to coding standards and inefficiencies, **minor, medium, and major flaws** were identified that had to be remediated as soon as possible to ensure the contracts of the Jasper Platform are of the highest standard and quality.

These inefficiencies and flaws **were swiftly dealt with by the development team behind the Amung project**, indicating that **the team possesses the necessary discipline and coordination** to develop code of the security standard expected from a blockchain project.

As a general comment, the use case itself relies on the presence and management of the contracts from an external party, the blockchain “Oracle” and the Jasper Trading Platform,

meaning that **the most stringent security standards should be imposed when handling the "managerial" private keys of the smart contracts** as they possess functionalities, such as the overriding of a previous "Order", that can lead to devastating results if maliciously used.

The test suites of the project were a plus and provided great insight to the inner workings of the contracts. They helped ensure that the smart contract code performs well under normal circumstances and depicted how they should be used in those circumstances.

To conclude, the review exposed certain issues of the project's codebase that **were alleviated and thus render the codebase compliant with the latest coding standards and practises.**

Recommendations

With regards to the new codebase, the main recommendation we can make is **the expansion of the test suites to encompass more edge cases** and ensure that the calculations done in the underlying functions that handle the "rounding" of numbers and the general wad-based math calculations are functioning as you expect.

The DSMath library utilized for wad-based math comes from a respectable figure in the Solidity development world, however it is not as battle-tested as OpenZeppelin's safe math libraries. To explicitly note, **the source code of the DSMath library used in the project was not audited** as it is not within the scope of or owned / developed by the project.

The following recommendations were fully taken into account by the Amun team and **remain here for historical purposes**.

Overall, we advise that the exhibits laid out in the report are carefully considered and actions are taken to resolve the issues that were identified. Each exhibit is associated with a section detailing the course of resolution where applicable and to the extent we can possibly advise.

The inconsistencies of the codebase are mostly a side-effect of developer preferences and as it appears multiple people have developed code on the smart contracts. A coding style should be enforced by an automatic tool such as a linter to ensure consistency throughout the project.

The test suites were descriptive; however we identified at least one that was incorrectly formed, namely L307 - L349 which tests for the failed redemption of a locked order. The mistake in this particular instance was the incorrect amount being attempted to be redeemed as the variable “tokensSent” should be swapped with “tokensCreated” due to the conversion.

Closer inspection of the test suites should be conducted to ensure that they fulfill their intended description correctly to avoid any incorrectly coded tests from “validating” a functionality that is otherwise faulty, as is the case with Exhibit 35.

An additional note, we advise the usage of some form of code coverage tool to aid in assessing whether the currently provided test suites correctly test the functionality of the smart contracts and help identify potential functions or code blocks whose functionality remains untested.

With regards to the code itself, the contract utilizes dynamic arrays in multiple segments of the codebase that may increase gas cost as dynamic arrays behave differently than mappings in their Read, Update & Delete operations.

Dynamic arrays are quite similar to mappings with regards to their layout, as they utilize a hashing function to map a position of the array to a position in the memory of the contract. The

main difference between the two is that the length of the array is stored as a variable in memory and updated during any operations that affect the length such as insertions and deletions.

This difference is negligible, as converting a dynamic array to a mapping would still need one to use a seperate variable for retaining the length. The main drawback of dynamic arrays is that they conduct redundant checks on “Read”, “Update” and “Delete” operations with regards to the length of the dynamic array. The codebase of the contract indicates that many functions already verify the index of the dynamic array to be within bounds and as such, it may improve gas efficiency by 200-300 gas per operation to convert the contracts to utilize mappings instead.

Overall, the codebase of the contracts should be refactored to assimilate the findings of this report, enforce linters and / or coding styles as well as correct any spelling errors and mistakes that appear throughout the code **to achieve a high standard of code quality and security**.

Findings (Original Version)

Exhibit 1

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Unsanitized Variable Multiplications and Divisions	Arithmetic Operations	Minor	CollateralPool.sol Line 59	CashPool.sol Line 66

[MINOR] Description:

The specified line is meant to calculate the percentage of the total amount moved to the pool that will be moved to the cold storage address. However, the variables of "percentageOfFundsForColdStorage" are not sanitized at construction time & at "setPercentageOfFundsForColdStorage", potentially leading to the contract becoming unusable.

As a tangible example, should the representation of the percentage exceed 100% the transfer call at L62 will always fail, f.e. (percentageNumerator = 10) & (percentageDenominator = 1). Refer to Test Case 1 for a reproducible example.

The severity of the vulnerability is minor as a simple re-configuration will alleviate the issue and it cannot result in any temporary side-effects.

Recommendations:

We propose the inclusion of an additional sanity check in the "initialize" and "setPercentageOfFundsForColdStorage" functions that compares

percentageOfFundsForColdStorage[0] against percentageOfFundsForColdStorage[1] and asserts that the former is less than the latter to prevent misconfiguration of the contract.

Alleviation:

A change was imposed inside the “setPercentageOfFundsForColdStorage” function that prevents the aforementioned issue and an additional check was imposed in the “initialize” function of the contract conducting the test we have defined in the “Recommendations” section.

Exhibit 2

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Inconsistent Function Modifier Call Signature	Coding Style	Informational	CollateralPool.sol Line 50	CashPool.sol Line 57

[INFORMATIONAL] Description:

The modifier “onlyOwnerOrTokenSwap” is used twice in the specified contract, once with parenthesis (L73) and once with not (L50).

Recommendations:

We advise that the parentheses are added to the call located at Line 50 to ensure consistency in the codebase as the other contracts utilize modifiers with parentheses. This inconsistency is also observed in the PersistentStorage.sol (L110, L163, L210, L298, L316) contract and should be corrected there as well.

Alleviation:

The issue was covered in full, following the parentheses modifier invocation coding style.

Exhibit 3

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Redundant “require” comparison	Ineffectual Code	Informational	CollateralPool.sol Line 55	N/A

[INFORMATIONAL] Description:

The “require” call located at Line 55 of CollateralPool.sol ensures that the balance that has been approved from the “whiteListedAddress” to the contract is less than or equal to the balance intended to move to the pool, otherwise the call throws with the error message “cannot move more funds than allowed”. This check is redundant as the ERC20 implementation that the InverseToken inherits from already ensures this exact condition and throws the error message “ERC20: transfer amount exceeds allowance”.

Recommendations:

We advise that the “require” call is removed to optimize the gas cost of the function as, even in the case the underlying InverseToken contract allows one to transact a balance greater than the allowed one, this would point to a much more important vulnerability in the token itself rather than the CollateralPool contract.

Alleviation:

The issue was covered in full, adhering to our advice of removing the “require” check.

Exhibit 4

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Redundant “require” comparison	Ineffectual Code	Informational	CollateralPool.sol Line 77	N/A

[INFORMATIONAL] Description:

The “require” call located at Line 77 of CollateralPool.sol ensures that the balance that will be transacted to the “destinationAddress” from the contract is less than or equal to the balance of the contract, otherwise the call throws with the error message “cannot move more funds than owned”. This check is redundant as the ERC20 implementation that the InverseToken inherits from already ensures this exact condition and throws the error message “ERC20: transfer amount exceeds balance”.

Recommendations:

We advise that the “require” call is removed to optimize the gas cost of the function as, even in the case the underlying InverseToken contract allows one to transact a balance greater than the owned one, this would point to a much more important vulnerability in the token itself rather than the CollateralPool contract.

Alleviation:

The issue was covered in full, adhering to our advice of removing the “require” check. We advise that the function “moveTokenfromPool” be renamed to “moveTokenFromPool” to conform to the camel-case function naming format existing throughout the rest of the contracts.

Exhibit 5

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Redundant “return” statement	Ineffectual Code	Informational	CompositionCalculator.sol Line 50	N/A

[INFORMATIONAL] Description:

The “return” statement located at Line 50 of CompositionCalculator.sol is redundant as named return variables are automatically returned once the function’s execution ends.

Recommendations:

We advise that the “return” statement is removed to increase the consistency and legibility of the codebase.

Alleviation:

The issue was covered in full, adhering to our advice of removing the “return” statement.

Exhibit 6

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Optimization of code block	Coding Style	Informational	CompositionCalculator.sol Line 85 - 91	CompositionCalculator.sol Line 84 - 89

[INFORMATIONAL] Description:

The current function checks whether the new account balance will be higher than the previous one and assigns a boolean literal to the variable “negative” while also calculating the absolute difference in each if clause.

Recommendations:

To make the function more readable, we advise the assignment of the greater-than-or-equal boolean operand (newAcountBalance >= _balance) to the “negative” variable and then the replacement of the operand in the if clause with the negative variable. This will result in a single boolean assignment and thus a minor optimization of the code as well.

Alleviation:

The issue was addressed in full in accordance with our recommendations by storing the comparison to an “isNegative” boolean variable.

Exhibit 7

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Reliance on a block timestamp	Miner Influence	Informational	CompositionCalculator.sol Line 402	CompositionCalculator.sol Line 425

[INFORMATIONAL] Description:

The number of days that have been passed since the last rebalance rely on the latest block timestamp to calculate the day difference between the current date and the last rebalance.

Recommendations:

The latest block timestamp can be influenced by the miners of the blockchain within a small margin and as such should be generally avoided when unnecessary (See SWC-116). The exploitation vectors of this function are inexistent as a whole day difference cannot occur due to malicious activity as per SWC-116, however care should be taken to its usage as the timestamp reported around the timestamp a day changes may affect the correctness of this function.

Alleviation:

This was merely a warning and as such no remediation action was necessary.

Exhibit 8

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Redundant “return” statement	Ineffectual Code	Informational	CompositionCalculator.sol Line 454	N/A

[INFORMATIONAL] Description:

The “return” statement located at Line 454 of CompositionCalculator.sol is redundant as named return variables are automatically returned once the function’s execution ends.

Recommendations:

We advise that the “return” statement is removed to increase the consistency and legibility of the codebase.

Alleviation:

The issue was covered in full, adhering to our advice of removing the “return” statement.

Exhibit 9

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Usage of Magic Numbers to differentiate functionality	Magic Numbers	Minor	PersistentStorage .sol Line 119 - 123	PersistentStorage .sol Line 129 - 141

[MINOR] Description:

The if-else code block located between those lines differentiates the functionality of pushing a new element to the array and editing an existing element of the array based on the index of the array. Although it is explained within the comments of the function appropriately, it is generally advisable to avoid the usage of magic numbers to differentiate the functionality of the function, as is the case here with the number “100000000”. Additionally, apart from preventing the order located at index “100000000” from being editable, any number that is below “100000000” but greater than “orderByUser[whitelistedAddress].length” will cause the function to throw.

Recommendations:

To better illustrate what the code of the function does, we advise the refactor of the code segment above to the assignment of “orderByUser[whitelistedAddress]” to a storage variable of the array after which its length is compared to the “orderIndex” variable and the corresponding action is carried out. A separate boolean value could be used instead which would be more explicit, however for gas optimization purposes the above paradigm could be utilized where any number above the length of the array causes a “push” instead of an overriding action.

Alleviation:

The issue was addressed in full in accordance with our recommendations and a designated boolean value named “overwrite” was added to the function signature to differentiate functionality.

Exhibit 10

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Usage of Magic Numbers to differentiate functionality	Magic Numbers	Minor	PersistentStorage .sol Line 172 - 176	PersistentStorage .sol Line 190 - 194

[MINOR] Description:

See Exhibit 9.

Recommendations:

See Exhibit 9.

Alleviation:

See Exhibit 9.

Exhibit 11

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Incorrect Documentation	Coding Style	Informational	PersistentStorage .sol Line 180 - 184	PersistentStorage .sol Line 197 - 199

[INFORMATIONAL] Description:

The documentation included in the lines between 180 and 184 of PersistentStorage.sol are intended to describe the “getOrder” function, however they are the same comments as the ones for the “setOrder” function described in Exhibit 10.

Recommendations:

We advise the removal of the comments or their adjustment to reflect the function they are intended to describe.

Alleviation:

The issue was addressed in accordance with our recommendations by updating the comment lines.

Exhibit 12

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Unnecessary Assignment of Storage Array to Memory	Ineffectual Code	Informational	PersistentStorage .sol Line 226 - 241	N/A

[INFORMATIONAL] Description:

The aforementioned lines of code are meant to return the locked order of a user by accessing the “lockedOrders” address mapping to struct array mapping and then accessing a specific index of the struct array. However, the code first assigns the struct array of the mapping to memory before accessing it and returning the members of the struct.

Recommendations:

We advise that the assignment to memory is removed, as this may not affect external view operations but will affect any on-chain calls to the “getLockedOrderForUser” function, such as the one located at L239 of TokenSwapManager.sol. Instead, a direct assignment of the “CreateOrderTimestamp” struct storage pointer at the specific location of the address mapping and index of the struct array can be made at L235 and the corresponding struct members can be returned, leading to less gas consumed when invoked on-chain as well as less memory consumed on the node the contract is executed on.

Alleviation:

Locking of orders was removed altogether from the final version of the contract as it was deemed irrelevant by the development team and as such this exhibit no longer holds true.

Exhibit 13

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Unnecessary Assignment of Storage Array to Memory	Ineffectual Code	Informational	PersistentStorage .sol Line 226 - 241	N/A

[INFORMATIONAL] Description:

The function is meant to return the array size of the locked orders array located at the address of the “lockedOrders” mapping. However, the statements within currently copy the array from storage to memory before returning the length.

Recommendations:

Consult Exhibit 12. Instead of storing the storage pointer and accessing the struct members, we advise that the length of the array located at the specified address of the mapping is returned directly in a single statement. As a general rule of the thumb, it is advisable to first copy a struct to memory before accessing its members only if the members accessed are 4 or more, as the cost of copying from storage to memory would offset the cost of conducting 4 independent storage load operations.

Alleviation:

Locking of orders was removed altogether from the final version of the contract as it was deemed irrelevant by the development team and as such this exhibit no longer holds true.

Exhibit 14

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Inefficient “delete” Operation	Ineffectual Code	Informational	PersistentStorage .sol Line 374	PersistentStorage .sol Line 376

[INFORMATIONAL] Description:

The function is meant to delete the last minting fee bracket and fee, however the “delete” operation located at line 374 leads to more gas cost on the long run as the change and add operations on the minting fee brackets always override the fee located at the “mintingFee” mapping. This means that if a minting fee bracket that was previously initialized at a specific value is removed and re-initialized, the total gas cost of the transactions would be more than simply overriding an already initialized slot.

Recommendations:

This optimization step depends on whether it is envisioned previously deleted minting fee brackets will be reused. Judging by the existence of the “changeMintingLimit” function, previously deleted minting fee brackets will potentially never be reused and as such, the delete operation should be left as is. This depends on what inputs to the minting fee brackets are expected.

Alleviation:

Judging by the unaltered codebase, the development team decided that previously deleted minting fee brackets will seldom be reused. As such, no remediation action was taken.

Exhibit 15

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Incorrect “delete” Operation	Ineffectual Code	Medium	PersistentStorage .sol Line 375	PersistentStorage .sol Line 377

[MEDIUM] Description:

The function is meant to delete the last minting fee bracket and fee, however the “delete” operation located at line 375 does not fulfill its intended purpose. “delete” operations on arrays simply zero out the storage space instead of reducing the size of the array and essentially removing the element. This means that subsequent checks carried out at L368 and L379 will pass for values that are simply greater than zero, breaking the order of the array and leading to incorrect brackets being applied.

Refer to Test Case 2 for a reproducible example.

Recommendations:

The “delete” operation should be replaced by the “--” unary operator on the “length” member of the array. Internally, Solidity will free up the storage space at “length - 1” and subsequently reduce the length of the array as intended. If the Solidity compiler version is to be upgraded to 0.6, a “pop()” operation on the array should be conducted instead as operations on the “length” member are disallowed in Solidity version 0.6.

Alleviation:

The issue was addressed in full in accordance with our recommendations and the statement was replaced by applying the decrement operator “--” on the “length” of the “mintingFeeBracket” array.

Exhibit 16

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Inefficient Search on Sorted Array	Ineffectual Code	Informational	PersistentStorage .sol Line 390 - 397	PersistentStorage .sol Line 406 - 424

[INFORMATIONAL] Description:

The function is meant to search the “mintingFeeBracket” array for the first bracket that will be greater than or equal to the input “cash” variable. However, the search itself is inefficient as the operations can be reduced in half due to the traits of the “mintingFeeBracket” array.

Recommendations:

The functions “addMintingFeeBracket” and “changeMintingLimit” conform to the model of the “Insertion Sort” algorithm, meaning that the “mintingFeeBracket” array will at all times be sorted in ascending order. As the values that are compared are numerical, it is possible to apply the “Binary Search” algorithm to the array and find the bracket in few operations. However, the “Binary Search” algorithm itself is inefficient in Solidity as recursive algorithms cost much more gas cost than iterative algorithms. As such, a combination of the two should be used.

We advise that the code is revised to carry out the first step of the binary search algorithm and compare the middle element of the “mintingFeeBracket” to the “cash” variable. The comparison will yield which half of the array the iteration should look up, thus reducing lookup time and gas consumption in half. This can be significant if numerous minting fee brackets are envisioned to be included.

On an additional note, care should be kept to the effect of the “mintingFeeBrackets” size in relation to any “getMintingFee” calls, if they are done on-chain. While unlikely, it is possible this call will become impossible to execute if the array contains numerous elements leading to its execution exceeding the gas limit.

Alleviation:

The issue was addressed in accordance with our recommendations and the first step of the binary search algorithm was applied to reduce the loop iterations.

.

Exhibit 17

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Confusing Variable Suffix	Coding Style	Informational	PersistentStorage .sol Line 340 - 343, 345 - 348 & more	PersistentStorage .sol Line 326 - 335, 338 - 342 & more

[INFORMATIONAL] Description:

In multiple segments of the PersistentStorage.sol contract codebase, the usage of the “PerToken” suffix is utilized although the variable it refers to is a singular numeric value. This naming convention is confusing as one would expect the “PerToken” to refer to either a mapping of token names to values or an array of token values.

Recommendations:

Judging from the codebase & after consultation with the Amun AG technical team, we advise this to be renamed to “latestTokenX” or simply “latestX” where “X” denotes what the prefix of the “PerToken” variables is.

Alleviation:

The issue was addressed by renaming the suffix from “PerToken” to “PerTokenUnit” and the development team explained that the “PerTokenUnit” represents the representation of a fractional value i.e. miles per hour (miles / hour). Our advice is to properly reflect this in the documentation of the contract in the form of comments to aid in readability.

Exhibit 18

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Double “Getter” Function for Same Variable	Ineffectual Code	Informational	PersistentStorage .sol Line 35, 355 - 357	PersistentStorage .sol Line 31, 352 - 354

[INFORMATIONAL] Description:

As observed in the lines above, the variable “managementFee” is simultaneously declared as “public” and has a user-defined getter function “getManagementFee”. “Public” variables have a getter automatically generated by the Solidity compiler and as such, introduce unnecessary code to your contract and increase the gas cost of deployment.

Recommendations:

We suggest either the usage of the compiler-generated getter or the user-defined getter and the removal of the opposite for optimization of the code’s compiled output.

Alleviation:

The issue was addressed in full by rendering the “managementFee” variable “private” and as such avoiding the generation of a compiler-defined getter.

Exhibit 19

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Unsanitized Addition of Numeric Variables	Arithmetic Operations	Minor	PersistentStorage .sol Line 360	N/A

[MINOR] Description:

The operation included in line 360 of the contract adds the values of “getLendingFee()” and “getManagementFee()” and returns the output. Both of these variables are user defined and as such may lead to an overflow.

Recommendations:

The probability of the addition overflowing is negligible as these numbers act as fees and, if used correctly, since they represent fees they will never overflow as their values will be relatively small. However, the potential for overflow does exist if these variables are maliciously or erroneously added to the blockchain by an external factor. As “getTotalFee” is not used in the codebase of the project, we propose the function is removed altogether and external solutions simply add the values returned by “getLendingFee()” and “getManagementFee()” off-chain using number libraries that can safely store the addition of those values which at maximum will be $2^{256} + 2^{256} - 2$.

Alleviation:

The issue was addressed in full by removing the function altogether and carrying the addition off-chain as advised.

Exhibit 20

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Usage of Numerical Value as Boolean	Coding Style	Informational	PersistentStorage .sol Line 12	PersistentStorage .sol Line 13

[INFORMATIONAL] Description:

The usage of a “uint256” variable in contrast to a “bool” costs significantly more gas to alter and consumes more storage space while disrupting the consistency of the codebase to utilize the “is” prefix in boolean values, such as in “isOwner()” or “isPaused()”.

Recommendations:

We advise the conversion of the variable to a “bool” variable and the change of the respective function signatures and function bodies it is involved in. As a side-effect, this will also lead to the variables “bridge”, “isPaused” and “isShutdown” being packed into a single 32 byte slot due to Solidity’s tight-packing mechanism and as such will optimize the gas cost of operations regarding these variables as well as reduce the deployment cost of the contract.

Alleviation:

The issue was addressed in full by changing the variable from a “uint256” to a “bool” as advised and having the change reflected wherever needed.

Exhibit 21

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Inconsistent Usage of “set” Prefix in Setter Functions	Coding Style	Minor	PersistentStorage .sol Line 75 - 77, 79 - 81	PersistentStorage .sol Line 73 - 75, 77 - 79

[MINOR] Description:

In the first instance of the “set” prefix, “setIsPaused”, the function assigns the input variable to the internal storage as expected of setter functions. In the second instance of the “set” prefix, “setIsShutdown”, the function assigns the numeric literal 1 (signaling “shutdown”) to the “isShutdown” variable and does not possess any inputs thus locking the contract permanently if called.

Recommendations:

We advise the “setIsShutdown” function be renamed to a more legible name, such as “shutdown()”, or, if we wish the function to not freeze the contract in place, we advise that an input variable is put in the function signature and properly utilized in the function body.

Judging by the codebase of TokenSwapManager, malicious or erroneous use of the “setIsShutdown” function can lead to funds being permanently freezed within the token pool unless actuated by the “owner” of the contract directly.

Alleviation:

The issue was addressed in full by renaming the second instance of the “set” prefix to simply the verb “shutdown”.

Exhibit 22

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Potential Optimization of “Accounting” Struct	Coding Style	Informational	PersistentStorage .sol Line 14 - 19	PersistentStorage .sol Line 15 - 20

[INFORMATIONAL] Description:

The “Accounting” struct, as is, occupies 4 32-byte slots within the blockchain. It should be possible to reduce this occupation to 3 32-byte slots, significantly reducing the gas cost of creating an “Accounting” struct entry on the blockchain by tight-packing the variables “cashPositionPerToken” and “lendingFee” into a single 32-byte slot.

The “cashPositionPerToken” and “lendingFee” should never exceed the size of 2^128 (340282366920938463463374607431768211456) a uint128 occupies, as it far exceeds the global economy’s value in any denomination (what the “cashPositionPerToken” should theoretically be capable of holding) as well as the value the “lendingFee” value is meant to hold at maximum judging by L59-L68 of CompositionCalculator.sol.

Recommendations:

These are merely estimations as we are not the creators of the codebase and we do not know the true value ranges these variables are anticipated to hold. What we explicitly advise is for you to evaluate the “Accounting” struct and, if you deem it possible, tight-pack its members to optimize gas costs.

Alleviation:

The optimization was taken into consideration and will be kept in mind for a future release of the project.

Exhibit 23

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Incorrect Usage of Prefix “set”	Coding Style	Informational	PersistentStorage .sol Line 204 - 219	N/A

[INFORMATIONAL] Description:

Here, the prefix “set” is used on the function name “setLockedOrderForUser” which internally pushes a new locked order to the array of locked orders of a specific user. The prefix “set” implies that a value is assigned whereas at this function a variable is inserted.

Recommendations:

We propose the renaming of the function to a more legible name, such as “insertLockedOrderForUser”, to optimize legibility and aid external readers of the ABI in utilizing the contract correctly.

Alleviation:

Locking of orders was removed altogether from the final version of the contract as it was deemed irrelevant by the development team and as such this exhibit no longer holds true.

Exhibit 24

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Inefficient Usage of Strings	Ineffectual Code	Informational	TokenSwapManager.sol	TokenSwapManager.sol

[INFORMATIONAL] Description:

It appears that throughout the codebase of TokenSwapManager strings are used alongside the imported “Strings” library inefficiently. The sole member of the library that is being utilized is the “compareTo” function, meaning that strings are used to categorize actions and / or assets, as evidenced by L81, L152 and L213. This type of categorization is inefficient as passing strings between contracts is cost-intensive and at times is unnecessary within the TokenSwapManager contract since all comparisons are done with string literals rather than dynamic variables.

Recommendations:

We advise the avoidance of the “Strings” library altogether and the mitigation of the string comparison requirement in the aforementioned lines.

In detail:

1. L66 “createOrder” & L132 “redeemOrder”: Here, the in-memory “result” string is only utilized to be compared to the string literal “ERROR” and revert the creation order if the comparison succeeds. Instead, this could be swapped with a simple boolean indicating that the call has errored.
2. L202 “writeOrderResponse”: Here, the string value of orderType is compared against the string literals “CREATE” and “REDEEM”. In general, across the whole codebase of the project the “orderType” member of the “Order” struct in PersistentStorage.sol[L26-L31]

could instead be converted to an enum. An “enum” will not only make the codebase more readable but would also render all operations conducted on the variable much more efficient. If these string literals are used by external services via the “getOrderByUser” and “getOrder” functions of the PersistentStorage contract, a straightforward literal assignment can be done by a chained if-else statement in the body of each function that converts the enum to the string literal desired.

Alleviation:

All types of string comparisons were removed from the contract and the Strings library was omitted to fulfill our recommendation.

Exhibit 25

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Unsanitized Initializer Input	Contract Freeze	Minor	TokenSwapManager.sol Lines 42 - 60	TokenSwapManager.sol Lines 42 - 70

[MINOR] Description:

Within the initializer of the contract, the addresses of the various contracts utilized are not compared against the “zero” address as done in the codebase of ColateralPool.sol[L19-L42] thus potentially locking the contract if the variables are not properly passed by whatever software is being utilized to interface with the blockchain. Additionally, the “CollateralPool” contract exposes the “kycVerifier” and “persistentStorage” addresses as they are declared as “public”, meaning that the initializer of TokenSwapManager can use those addresses instead of relying on input variables.

Recommendations:

We propose that the addresses of “_stablecoin”, “_inverseToken”, “_collateralPool” and “_compositionCalculator” are compared against zero to ensure that no erroneous usage of the constructor will be fulfilled and we also advise the addresses already exposed by the contract of “CollateralPool” are utilized, optimizing the gas efficiency of the initializer function and ensuring code-based consistency of addresses.

Alleviation:

The issue was addressed in full by carrying out the comparisons that we named above as well as retrieving the pre-existing data from the “CollateralPool” contract, now renamed to “CashPool”.

Exhibit 26

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Usage of Functions Always in Sequence	Ineffectual Code	Informational	TokenSwapManager.sol Lines 95 - 109, 170 - 184	TokenSwapManager.sol Lines 109 - 117, 167 - 175

[INFORMATIONAL] Description:

It appears that the functions “setOrderByUser” [101-124] and “setOrder” [155-178] of PersistentStraoge.sol are never used independently. This means that those functions could be combined in some way to fulfill both functionalities with a single endpoint.

Recommendations:

We propose that “setOrderByUser” internally calls “setOrder” when a new order is being created or, to optimize the process, a single function is called that fulfills the functionality of both “setOrderByUser” and “setOrder” to avoid the surplus comparisons being done on both.

Additionally, the functions themselves appear to be relatively ambiguous. The “set” prefix implies that the functions assign a value to a variable whereas the usage of the magic number “100000000” per Exhibits 9 and 10 actually appends new data to the variable. This should be avoided and separate functions should be utilized for appending and overriding, if at all necessary.

Alleviation:

The issue was addressed in full by refactoring the “setOrderByUser” function to internally call the “setOrder” function.

Exhibit 27

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Temporary Out-of-Gas Vulnerability	Contract Freeze	Minor	TokenSwapManager.sol Lines 226 - 245	N/A

[MINOR] Description:

The “getLockedAmount” function can potentially lock the execution of “redeemOrder” [L132-L200] as its gas cost increases linearly with the number of orders created for a specific address. In detail, the function initially retrieves the number of locked orders. The locked orders increase as an address transacts with the “TokenSwapManager” and creates a new order, meaning that this array is anticipated to increase in size relatively quickly.

Afterwards, a loop is conducted that begins at the array’s size and “loops” through the locked orders until it finds one that has been conducted more than an hour ago and breaks the loop, resuming the execution of “redeemOrder”.

If multiple orders are created in a short period of time, the function will temporarily be blocked as the “getLockedOrderForUser” function is relatively gas expensive due to it accessing a mapping, transferring data from storage to memory and passing it back to the original caller.

Recommendations:

No apparent mitigation exists as this vulnerability exists by design of what the contract is meant to accomplish but care should be taken to what the temporary locking affects and software-side

mitigations should exist e.g. proper error-catching. Additionally, certain optimizations can be conducted.

In its current form, it conducts a subtraction on each loop iteration at L239 whereas this would be unnecessary if the loop itself instead began at “count - 1” and continued until zero. Furthermore, the last return statement is unnecessary as the return variable has been declared and properly assigned the return output.

Alleviation:

Locking of orders was removed altogether from the final version of the contract as it was deemed irrelevant by the development team and as such this exhibit no longer holds true.

Exhibit 28

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Exploitable “require” Logic	Ineffectual Code	Major	TokenSwapManager.sol Lines 142 - 145	N/A

[MAJOR] Description:

It appears this “require” statement verifies that the latest locked amount of an account does not exceed the tokens that are to be redeemed. This appears to be logically incorrect, however, as orders are atomic and as such, a previous deposit of 5 tokens that may have passed the 1 hour threshold would not be redeemable if a new order was made that is below that amount. This is a consequence of Orders not being interconnected with identifiers and instead being sequentially added to the blockchain.

Additionally, as locked orders are not affected when a “REDEEM” operation is conducted, it is possible to withdraw the locked amount in two transactions instead of attempting to withdraw the whole amount on one transactions as the logical comparison of those lines is that of a “less-than” operation that is satisfiable if the locked amount is broken up to f.e. “amount - 1” and “1”.

Refer to Test Case 3 for a reproducible example.

Recommendations:

A mitigation would be to somehow associate orders with a particular ID, however this would require restructuring of the contract in multiple places and would not be advisable at this stage

unless sufficient time and resources are allocated. What we advise is the careful investigation of what the aftereffects of this “require” statement are and what actions it would prohibit in your actual software implementation.

Additionally, a solid solution to the segmented-transactions exploit should be applied as this can potentially break your application and allow someone to extract the full locked token amount in two transactions with carefully devised transactions. Both of these issues would be solved with using identifiers for orders.

Alleviation:

Locking of orders was removed altogether from the final version of the contract as it was deemed irrelevant by the development team and as such this exhibit no longer holds true.

Exhibit 29

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Inefficient “require” Statements	Ineffectual Code	Informational	TokenSwapManager.sol Lines 202 - 224	TokenSwapManager.sol Lines 193 - 196

[INFORMATIONAL] Description:

The “require” statements of this function are either ineffective as they are already guaranteed by other segments of the codebase or inefficient as they conduct unnecessary comparisons.

The first “require” statement of the function body conducts a “greater-than” comparison of unsigned integers against zero. As unsigned integers cannot be “less-than” zero, it is more gas efficient to instead simply conduct a “not-equal-to” operation.

The second “require” statement checks that the “orderType” to be emitted in the event is either “CREATE” or “REDEEM”. As this function is “internal”, this is already guaranteed by the locations the function’s calls exist as they pass the string literals “CREATE” and “REDEEM” on lines 118 and 188 respectively.

The third “require” statement ensures the “whiteListedAddress” is not equal to the “zero” address, 0x0...0. This is guaranteed by the fact that it is a whitelisted address, as a check already exists on L260 of PersistentStorage.sol and as such, this comparison is unnecessary.

Recommendations:

We propose the removal of the latter two “require” statements and the optimization of the first one as per the description laid out above.

Alleviation:

The issue was addressed in full by following our recommendations to the letter.

Exhibit 30

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Inconsistent Specifier of “Ownable” Initializer	Coding Style	Informational	CollateralPool.sol Line 37 PersistentStorage .sol Line 55 TokenSwapMana ger.sol Line 51	CashPool.sol Line 36 PersistentStorage .sol Line 53 TokenSwapMana ger.sol Line 50

[INFORMATIONAL] Description:

The “initialize” function of the “Ownable” contract is called in all of the lines specified above, however the contract specifier in some lines exists whereas on the others does not.

Recommendations:

To ensure consistency in the codebase, we suggest that the specifier is either included in L37 of CollateralPool & L51 of TokenSwapManager or removed from L55 of PersistentStorage, the former of which we recommend.

Alleviation:

The issue was addressed in full by removing the contract specifier in all “initialize” calls.

Exhibit 31

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Usage of Both “now” and “block.timestamp”	Coding Style	Informational	CompositionCalculator.sol Line 402 PersistentStorage.sol Line 300 TokenSwapManager.sol Line 113, 240	CompositionCalculator.sol Line 425 PersistentStorage.sol Line 271

[INFORMATIONAL] Description:

In the lines mentioned above, the unix timestamp of a block is inferred from either the global variable “now” or from the global variable “block.timestamp”.

Recommendations:

As “now” is a direct alias of “block.timestamp”, we propose the usage of either “now” or “block.timestamp” uniformly across the codebase, the latter of which we advise as it is more explicit.

Alleviation:

The issue was addressed in full by using “block.timestamp” throughout the contract instead of “now”.

Exhibit 35

TITLE	TYPE	SEVERITY	LOCATION (OLD)	LOCATION (NEW)
Invalid Mathematical “power of” Operation	Arithmetic Operations	Medium	PersistentStorage .sol Line 62, 364, 396	PersistentStorage .sol Line 60, 358, 423

[MEDIUM] Description:

The aforementioned lines contain the “2^256-1” numeric literal. At first glance, this is meant to raise 2 to the power of 256 and subtract the numeric literal 1 from the final result. Apart from the intermediary value overflowing (2 to the power of 256 would exceed the maximum capacity of 2 to the power of 255 minus 1), the mathematical symbols utilized actually carry out a bitwise exclusive OR operation (XOR), as the “power of” operator is signified by two consecutive asterisks “**” in both Solidity and JavaScript. As such, the bracket “253” is used in the mapping as the “maximum” value bracket as bitwise operators are conducted last in Solidity. This can lead to an invalid fee being set if a minting fee bracket is equal to “253” as well as obfuscates the actual result of the code versus the apparent intention.

Recommendations:

The code should be refactored to properly depict the maximum “uint256” value as a literal. This can be done by a bitwise negation operation on the numeric literal 0: “~uint256(0)”

Additionally, it appears that the test case of the PersistentStorage contract (PersistentStorage.test.js) utilizes the same bitwise exclusive OR operation on line 344. This should be refactored as well, as the incorrect validity of the test would imply the latest fee bracket is properly assigned at the numeric literal key “253”.

Alleviation:

The issue was addressed in full by using the “~uint256(0)” format of specifying the maximum unsigned integer possible in Solidity. We propose the storage of the said variable as a “constant” contract variable to something akin to “MAX_SAFE_UINT” to aid the readability of the code. “constant” variables are not actually included in the compiled bytecode as actual contract variables and are simply replaced wherever they are mentioned during compilation so gas cost will remain unaffected.

Findings (Revised Version)

Exhibit 1

TITLE	TYPE	SEVERITY	LOCATION
Double “Getter” Function for Same Variable	Ineffectual Code	Informational	PersistentStorage .sol Line 43, 101 - 107

[INFORMATIONAL] Description:

As observed in the lines above, the variable “delayedRedemptionsByUser” is simultaneously declared as “public” and has a user-defined getter function “getDelayedRedemptionsByUser”. “Public” variables have a getter automatically generated by the Solidity compiler and as such, introduce unnecessary code to your contract and increase the gas cost of deployment.

Recommendations:

We suggest either the usage of the compiler-generated getter or the user-defined getter and the removal of the opposite for optimization of the code’s compiled output.

Alleviation:

The issue was addressed in full in update 55e2da379a7ba46a882a7e32ae259d796ed4aa28 of the project’s repository by removing the “Getter” function and favoring the compiler-generated one.

Exhibit 2

TITLE	TYPE	SEVERITY	LOCATION
Confusing Event Input	Coding Style	Informational	TokenSwapManager.sol Line 264

[INFORMATIONAL] Description:

The “writeOrderResponse” function invoked at the specified line emits a “SuccessfulOrder” event, the first argument of which is supposed to be the “orderType” string, in our case the “REDEEM NO SETTLEMENT” string. Judging by the codebase, this type order is meant to be settled later on when the hot wallet funds of the contract have been sufficiently supplied with the necessary token. This is not apparent by the “orderType” naming, as “NO SETTLEMENT” points towards no settlement taking place.

Recommendations:

We suggest the usage of “REDEEM PENDING SETTLEMENT” instead of “REDEEM NO SETTLEMENT” as it is more descriptive with regards to what it entails.

Alleviation:

The issue was discussed with the Amun AG team and we concluded that the naming of the event makes sense as the event’s emittance does not guarantee that a settlement will occur in the future. On an additional note, the team chose to prefer an underscore naming convention rather than a space one to avoid any potential issues with the space character and this was addressed in full in update 55e2da379a7ba46a882a7e32ae259d796ed4aa28 of the project’s repository.

Exhibit 3

TITLE	TYPE	SEVERITY	LOCATION
Unprotected Function Invocation	Access Control	Minor	TokenSwapManager.sol Line 206 - 234

[MINOR] Description:

The delayed redemption of funds function “settleDelayedFunds” is not protected by the “notPausedOrShutdown” modifier in contrast to how other functions of the contract are handled and as such can be executed when the contract is paused and / or shutdown.

Recommendations:

We suggest the inclusion of the modifier in the function’s declaration to ensure that contract pausing and shutdown works as intended.

Alleviation:

The issue was addressed in full in update 55e2da379a7ba46a882a7e32ae259d796ed4aa28 of the project’s repository by invoking the modifier correctly in the function signature.

Test Cases (Original Version)

The test cases below were meant to illustrate the vulnerabilities that were exposed in the initial version of the report and as such will not function in the new version. They remain for historical purposes.

Test Case 1

Description:

This test case is meant to replicate the issue identified on Exhibit 1. It should be included in the test file for the CollateralPool contract (CollateralPool.test.js at Line 215) and attempts to conduct a transaction after incorrectly setting the “percentageOfFundsForColdStorage” variable to invalid amounts. Code is located on the next page.

```
// Code Segment 1
it("does NOT allow to set incorrect values", async function() {
  await expectRevert(
    this.contract.setPercentageOfFundsForColdStorage([2, 1], {
      from: owner
    }),
    "incorrect percentage provided"
  );
});
// The test below should FAIL
it("currently throws if incorrect values are set", async function() {
  await this.contract.setPercentageOfFundsForColdStorage([2, 1], {
    from: owner
  });

  await token.approve(this.contract.address, amountOfTokensToPool, {
    from: user
  });

  await expectRevert(
    this.contract.moveTokenToPool(
      token.address,
      user,
      amountOfTokensToPool,
      {
        from: owner
      }
    ),
    "ERC20: transfer amount exceeds balance"
  );
});
```

Test Case 2

Description:

This test case is meant to replicate the issue identified on Exhibit 15. It should be included in the test file for the PersistentStorage contract (PersistentStorage.test.js after Line 388) and replicates the issue where an incorrect minting fee bracket can be inserted after deletion of the latest bracket. Code can be found below:

```
// Code Segment 2
// The BN member of the @openzeppelin/test-helpers package should be
properly imported
it("properly deletes minting fee disallowing fees less than the highest
one", async function() {
  await this.contract.deleteLastMintingFeeBracket({ from: owner });
  const highestMintingFeeLimit =
    await this.contract.mintingFeeBracket("0");
  await expectRevert(
    this.contract.addMintingFeeBracket(
      highestMintingFeeLimit.sub(new BN(1)),
      "1",
      {
        from: owner
      }
    ),
    "New minting fee bracket needs to be bigger then last one."
  );
});
```

Test Case 3

Description:

This test case is meant to replicate the issue identified on Exhibit 31. It should be included in the test file for the TokenSwapManager contract (TokenSwapManager.test.js on line 350) and replicates the issue where the full locked order amount can be retrieved via two separate carefully crafted transactions. The code can be found on the next page.

```
// Code Segment 3
it("locks tokens from multiple creation orders even if requested
separately", async function() {
  const tokensSent = getEth(10);
  const tokensCreated = getEth(0.00997);
  const orders = 3;

  for (let i = 0; i < orders; i++) {
    await this.tokenSwapManager.createOrder(
      "SUCCESS",
      tokensSent,
      tokensCreated,
      2,
      price,
      user,
      { from: bridge }
    );
  }

  const lockedTokens = await this.tokenSwapManager.getLockedAmount(user, {
    from: user
  });

  expect(lockedTokens.toString()).to.be.equal(
    new BigNumber(tokensCreated).times(orders).toString()
  );

  const compositionForN =
    await this.compositionCalculator.getCurrentCashAmountCreatedByToken(
      new BigNumber(tokensCreated).times(orders),
      price
    );

  const compositionForSingle =
    await this.compositionCalculator.getCurrentCashAmountCreatedByToken(
      new BigNumber(tokensCreated),
      price
```

```
  );

const compositionForMinusOne =
  await this.compositionCalculator.getCurrentCashAmountCreatedByToken(
    new BigNumber(tokensCreated).times(orders).minus(1),
    price
  );

await expectRevert(
  this.tokenSwapManager.redeemOrder(
    "SUCCESS",
    new BigNumber(tokensCreated).times(orders),
    compositionForN,
    2,
    price,
    user,
    { from: bridge }
  ),
  "cannot redeem locked tokens" // From all orders
);

await expectRevert(
  this.tokenSwapManager.redeemOrder(
    "SUCCESS",
    new BigNumber(tokensCreated),
    compositionForSingle,
    2,
    price,
    user,
    { from: bridge }
  ),
  "cannot redeem locked tokens" // From one order
);

await expectRevert(
  this.tokenSwapManager.redeemOrder(
    "SUCCESS",
```

```
      new BigNumber(tokensCreated).times(orders).minus(1),
      compositionForMinusOne,
      2,
      price,
      user,
      { from: bridge }
    ),
    "cannot redeem locked tokens" // From all orders minus 1
  );
});
```