BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT:

usual.money

February 2024

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Usual Money
Website	usual.money
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/usual-dao/pegasus/tree/3fb5b4910a2131d052e49ffc8bb31f9969b85597/packages/solidity/src
Resolution 1	https://github.com/usual-dao/pegasus/tree/9e42b9590bf62373d066245ad5a35f8177d1f375/packages/solidity/src

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	27	17	5	5
Medium	21	8	1	12
Low	48	24	2	22
Informational	41	18	1	22
Governance	5	5	0	0
Total	142	72	9	61

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection Overview

Disclaimer: This audit does not include any potential issues which can arise due third-party provider integrations such as these from RWA-Providers.

Security History:

Pre-Audit Phase

Prior to our comprehensive audit, the codebase was subjected to Spearbit's vCISO service (https://hackmd.io/@spearbit/BJXzIlwTq) , incorporating one Lead Security Researcher (LSR) to guide through smart contract best practices, perform an architectural review, and support the development framework. This preliminary stage aimed at preparing the codebase for an in-depth audit, ensuring it was primed for a thorough examination.

Audit Findings and Refactoring Plan

Throughout the auditing process, we identified a significant number of issues:

27 High

21 Medium

48 Low

41 Informational

5 Governance

Given the extensive range of critical and non-critical issues uncovered, a straightforward resolution process was deemed impractical. Consequently, a comprehensive refactoring of the codebase has been necessitated. We have allocated a 14-day period, from February 19th to March 3rd, dedicated to post-refactoring review. During this interval, one sole senior auditor will oversee the examination of the codebase and the implementation of fixes. It is imperative to acknowledge that by the conclusion of this period, the codebase will not be ready for deployment, potentially harboring multiple vulnerabilities. This limitation stems from the constrained review timeframe and the feasibility of a single auditor fully addressing all issues, especially considering the initial discovery of 96 (High-Medium-Low) bugs.

The objective of this phase is to elevate the codebase to a state conducive for a Spearbits audit, enabling Spearbits engineers to apply advanced exploit detection methodologies.

Final Audit by Spearbit

Commencing on March 4th, Spearbit will undertake the final audit phase for this codebase. This stage represents the culmination of our audit efforts, aiming to rigorously assess and fortify the codebase against vulnerabilities. Through Spearbit's expertise in advanced exploit detection, this final audit seeks to ensure the highest level of security and readiness for the codebase.

Auditor Assignment:

This initial audit was conducted by one senior auditor and one trainee. This was communicated with the usualUSD team and reflected by the cost.

Global:

The Global section of this audit report is dedicated to identifying and analyzing issues that are not confined to a single contract or function but rather span across the entire codebase or affect multiple contracts within the project. This section is crucial for understanding systemic risks, overarching design patterns, and consistency issues that might not be apparent when examining individual contracts in isolation.

Issue	General Governance Privileges can result in a full loss of funds
Severity	Governance
Description	The full architecture assumes that the governance body is trusted. In a scenario of a compromised governance body this can result in a total loss of funds due to various mechanisms, the most notable abuse would be manipulating the RegistryAccess to grant several minter roles or the draining of the underlying RWA assets. Several other privileges that can result in a full DoS or loss of all assets are spread throughout the codeba
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged

Issue	Multiple contracts do not comply with RegistryContract changes
Severity	Low
Description	The RegistryContract keeps track of all corresponding contract addresses and allows the usualTech address to change these. The problem with this architecture is that many contracts, such as example the BondDistributor contract fetch the addresses upon

	deployment and then hardcodes these. This will not work if at any point the contracts in the registry are changed.
Recommendations	Consider if that will expose a problem, if yes consider implementing setter functions for all contracts which can update the specific addresses based on the registry entry.
Comments / Resolution	Acknowledged

Issue	**Architecture is not developed to support multiple STBC**
Severity	Informational
Description	While the contract incorporates factories and certain mechanisms indicate that it might be desired to have multiple STBCs, certain spots within the codebase would make this impossible, such as the following: address tokenToMint = _tokenMapping.getStbcFromRwa(rwaToken); This will obviously not work for multiple STBCs since a RWA can only have one STBC assigned and therefore the same RWA cannot be used for multiple STBCs. Moreover, the team indicated that in V0 this is not desired.
Recommendations	Consider refactoring the codebase for V1 in an effort to support multiple STBCs, multiple spots in the codebase would need to be adjusted.
Comments / Resolution	Acknowledged

Issue	Decimal normalization can result in truncation
Severity	Informational
Description	Throughout the codebase, decimals are adjusted to 18, which serves as a standard to ensure correctness for arithmetic operations. For tokens with more than 18 decimals, this means that the amount will be truncated, resulting in a slightly lower value than originally provided.
Recommendations	We do not necessarily see an issue here, however, for consistency purposes it is still mandatory to report it to the client.
Comments / Resolution	Acknowledged

Issue	Protocol does not work with transfer-tax tokens
Severity	Informational
Description	Even though the protocol is not intended to work with transfer-tax tokens, it is still necessary to formally report the fact that this architecture cannot handle such tokens.
Recommendations	Consider not adding such tokens.
Comments / Resolution	Acknowledged

BucketDistribution.sol

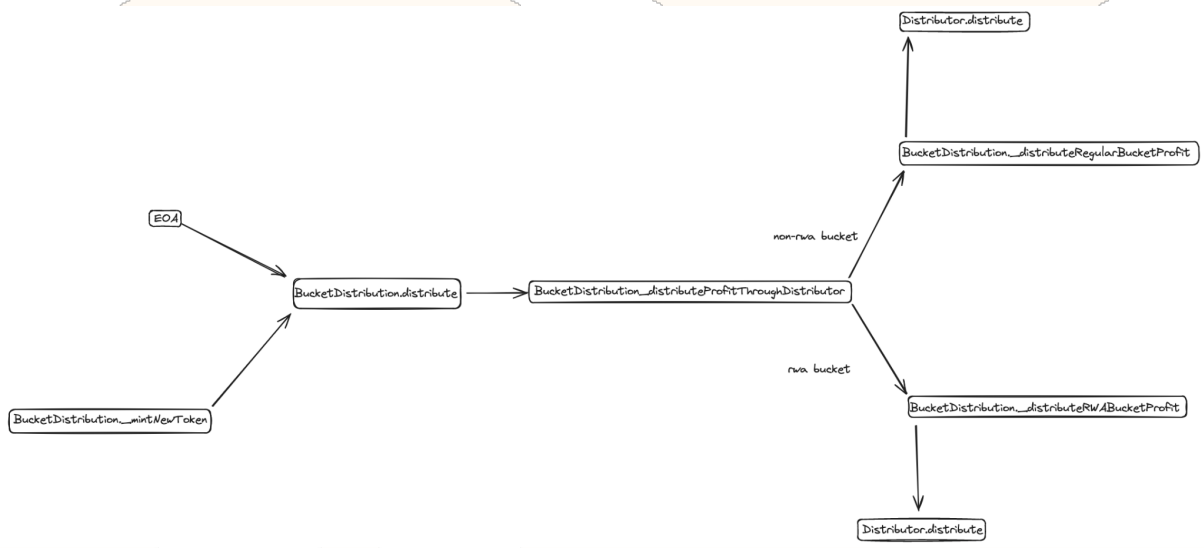
The BucketDistributor contract is the heart of the bucket module and facilitates the core functionality for buckets. On a high-level description of the governance functionalities, the DAO address can create two specific types of buckets:

1. Non-RWA Bucket: This bucket is not correlated with a stablecoin nor an RWA asset and simply fungates as a pass-through bucket which accumulates fees from specific purposes in the _balance state variable. On different occasions including a simple EOA-call, these fees are then distributed via a distribution call-path towards a recipient.

2. RWA-Bucket: This bucket is correlated with a stablecoin and allows for one or more RWA's backing the stablecoin. Upon the minting process, users deposit RWA assets into the contract and receive the equivalent of the stablecoin. The RWA asset simply stays in the bucket contract and solely serves as backing for the stablecoin, which makes it trivial for users to inspect if the stablecoin is in fact sufficiently backed. For the scenario that the RWA appreciates in value, this appreciation delta is taken as profit for the protocol, which is then also transferred through a distribution call-path. Additionally to the stbc minting for the profit, a specific amount of usDAO is minted as well, depending on the usdaoEmissionRate.

While the codebase indicates that there can be multiple RWA-buckets and multiple STBCs, the team mentioned that for the current v0, there will only be one stablecoin and therefore one RWA-bucket. Nevertheless, in an effort to contribute to a safer web3 landscape, we aim to aggregate all potential issues which could arise due to the implementation of multiple RWA-buckets, respectively multiple stablecoins

The aforementioned distribution flow is illustrated as follows:



The overall architecture indicates that buckets for the following tokens will be created:

Non-RWA buckets:

1. USDT (minterOffer cancellation)

2. USDC (minterOffer cancellation)

3. LsausUSD (transfer-tax)

4. STBC (redeem fee)

5. usDAO (mint reward)

RWA buckets:

1. RWA (STBC backing via normal mint or MintEngine execution)

Threat Modeling:

The BucketDistribution contract mainly involves configurational and accounting risk, since the way for users to *directly interact* with the contract is limited to the mintNewToken and distribute functions. Moreover, additional risk is the lack of validation from third-party contracts which can manipulate the BucketDistribution state via cross-contract interactions.

Potential Risk Parameters (PRP)

- Incorrect creation of buckets which can lead to accounting errors and loss of funds (permanent lock).

- Incorrect manipulation of settings via (for business logic redundant) functions, which can lead to accounting errors and loss of funds.

- Approval risks for distributors.

- Accounting errors to external manipulations from cross-contract entries, main target is the addAssetToBucket and burnAsset function.

- Unexpected side-effects of bucket state manipulation.

Issue	Removal of RWA from bucket results in loss of balance
Severity	**High**
Description	The removeRWATokenFromBucket function allows the DAO address to remove an RWA from their corresponding bucket. However, the storage delete will result in a loss of the balance: delete _rwaInfo[bucket][rwa]; which can therefore never be withdrawn.
Recommendations	Urgently consider checking if the RWA has an existing balance before removing it, if there is a non-zero balance, do not allow for removal.
Comments / Resolution	Not resolved. This will furthermore not only result in stuck funds but will also have an impact on the cbrCoef calculation, as these funds won't be part of the calculation anymore, which essentially results in a STBC depeg.

Issue	Users can grief burnAsset via a simple distribute call
Severity	**High**

Description	The burnAsset function is invoked upon the daoTokenUnwrap function within the DaoCollateral contract. To understand any potential issues, we first need to explain the rationale behind the daoTokenUnwrap function:
	
	This function simply allows the DAO address to burn an arbitrary amount of LusDAO tokens within the LusDAO bucket and mints the exact same amount of usDAO to the bucket, which can simply be seen as “unlocking” LusDAO:
	
	//remove amount of lusdao from balance
	_buckets.burnAsset(lusDAObucket, lusdao, amount);
	// mint usdao
	_usdao.mint(address(_buckets), amount);
	_buckets.addAssetToBucket(usDAObucket, usdao, amount);
	
	However, the achilles heel of this function lies within the fact that any bucket has a corresponding distributor. In our scenario the LusDAOBucket can be emptied upon two scenarios:
	
	a) Standard distribution mechanism which calls the corresponding distributor and transfers the LusDAO to a recipient.
	
	b) Unlock mechanism which is triggered by the daoTokenUnwrap function.
	
	Therefore, a malicious user can constantly call the distribute function, which will then permanently empty the balance of the bucket:
	
	if (profitToDistribute == 0) {
	
	profitToDistribute = _balance[bucket];
	}
	if (profitToDistribute != 0) {
	uint256 balanceBefore =

	IERC20(underlyingToken).balanceOf(address(this));
	profitedDistributed = distributor.distribute(
	bucket, underlyingToken, profitToDistribute, bondDistributorParam
	);
	_balance[bucket] -= profitedDistributed;
	
	This can be permanently abused to grief any daoTokenUnwrap call, effectively limiting the protocol's desired functionality and preventing unwrapping LusDAO to usDAO. This will essentially mean the locked usDAO token cannot be converted to unlocked usDAO.
	
	Additionally to this issue, it has to be emphasized that the only entrypoint for the burnAsset function is the daoTokenUnwrap function within the DaoCollateral contract, which explicitly checks that LusDAO bucket is not collateralized:
	
	if (_buckets.isBucketCollateralized(lusDAObucket)) {
	revert InvalidToken(); }
	
	However, the burnAsset function still allows the asset to be a potential RWA-asset, as it explicitly exposes a call-path for that scenario:
	
	// is it a collateralized bucket meaning token is actually a RWA
	if (_rwaInfo[bucket][token].isCollateral) {
	// it is a collateralized bucket so we need to remove to the rwa balance
	uint256 currentPrice = mintNewToken(bucket, token);
	_rwaInfo[bucket][token].balance -= amount;
	_rwaInfo[bucket][token].stbcBacked -= amount * currentPrice / SCALAR_ONE;
	}
	
	this call-path is therefore never triggered as there will never be a RWA asset parameter for the burnAsset function

Recommendations	First and foremost the distribute function should not be publicly

	callable to prevent any users from griefing. If desired that this function is periodically called by a keeper, it should be ensured that the keeper has a specific role to call this function. Moreover it must be noted that the keeper should then never call the distribute function for the LusDAOBucket. Additionally, consider if it might make sense to simply remove the RWA call-path from the burnAsset function, since as explained, it is redundant.
Comments / Resolution	Resolved, the distribute call is now guarded, moreover, it is now only possible to burn assets from regular buckets.

Issue	RWA's can be added to non-rwa bucket
Severity	Medium
Description	The addRWATokenToBucket function allows the DAO address to add an RWA to an arbitrary bucket, expected the bucket does not yet have the RWA assigned. However, there is no check that the bucket is in fact a RWA-bucket, which means that an RWA can also be added to a non-RWA-bucket. This would then result in undesired state transitions such as users potentially adding RWAs to the non-RWA-bucket. Moreover, there is no check if the provided bucket name in fact is an existing bucket, which could result to adding a RWA to a completely non-existent bucket.
Recommendations	Consider explicitly distinguishing between RWA and non-RWA-buckets to prevent such a situation.

	Consider validating that the bucket in fact exists (as RWA bucket), checking for _stbcInfo[bucket].token to be non-zero.
Comments / Resolution	Resolved, the function has been renamed to addRWATokenToUsUSDBucket and a check for _stbcInfo[bucket].token to be usUSD is enforced.

Issue	Same RWA for multiple buckets or simple donation will have an impact on the balance check
Severity	Medium
Description	During the mintNewTokens function, the balanceOf function on the RWA is used to determine if the function should return early. This is an issue if the same RWA is added to more than 1 RWA-buckets or if a user donates RWA towards the contract. This can result in unexpected state transitions.
Recommendations	A fix for this issue is non-trivial, we highly recommend the usualTech team to further elaborate this issue and potential fix scenarios.
Comments / Resolution	Resolved, this check has been removed, the rwaInfo.stbcBacked check is sufficient.

Issue	Loss of un updated profit during emergencyRWAWithdraw/deleteRWABucket
Severity	Medium

Description	The emergencyRWAWithdraw and deleteRWABucket function allows for withdrawing the profit up to the current state:
	
	// it is a rwa collateralized bucket
	// we transfer the underlying token
	StbcInfo storage stbcInfo = _stbcInfo[bucket];
	address underlyingToken = stbcInfo.token;
	uint256 profitBalance = stbcInfo.profitBalance;
	if (profitBalance != 0) {
	stbcInfo.profitBalance = 0;
	IERC20(underlyingToken).safeTransfer(safeAccount, profitBalance);
	}
	
	Which is perfectly fine to secure the accumulated profit. However, the profit is *not* updated before via mintNewToken. In a scenario where the RWA asset has appreciated in value, this appreciation is not catered during these calls.
	
	While we acknowledge that it might be desired to ignore the potential profit in an emergency situation, simply to limit the executed state transitions and therefore potential points of failure, we are still of the opinion to at least make it optionally for the caller to catch any additional profit, especially for the delete situation.

Recommendations	Consider implementing for the emergencyRWAWithdraw and deleteRWABucket an additional parameter which, if set to true, invokes the mintNewToken function.
Comments / Resolution	Acknowledged, no change was made.

Issue	Removal of STBC will brick bucket functionality
Severity	Medium

Description	The DAO address can remove the stbtc storage for any arbitrary bucket and its corresponding stbtc via deleteStbtcInfo:
	
	// transfer the profit balance to the safe
	uint256 profitBalance = _stbcInfo[bucket].profitBalance;
	_stbcInfo[bucket].profitBalance = 0;
	IERC20(stbc).safeTransfer(msg.sender, profitBalance);
	delete _stbcInfo[bucket];
	
	The deletion means that the:
	
	a) token
	b) profitBalance
	
	are zero'd out, remember, a mapping can still be accessed if it is deleted.
	
	Now during the normal business logic the mintNewToken function is invoked, which then specifically accesses the deleted mapping:
	
	StbcInfo storage stbcInfo = _stbcInfo[bucket];
	address underlyingToken = stbcInfo.token;
	
	whereas the underlying token is address(0).
	
	This storage pointer is then forwarded to the internal _mintNewToken function, which then triggers the mint:
	
	IStbc(underlyingToken).mint(address(this), amount);
	
	However, since the underlyingToken is address(0), this means the mint call will revert, effectively bricking the whole functionality, *irreversibly*. The only move-forward possibility is then to call emergencyRWAWithdraw or deleteRWABucket.

	Moreover, the removal of the stbcInfo, opens up the possibility for calling createRWABucket with an existing bucket, which opens up the possibility to override a distributor without revoking the approvals to the existing distributor. This is an important invariant that should hold. This should not be something that is possible in our opinion.
Recommendations	Consider simply removing this function and incorporating the stbtc deletion to the corresponding deleteRWABucket call. At the end there should be one single call which withdraws all corresponding RWA's for a bucket, deletes their storage as well as clears the stbc and distributor storage for one bucket.
Comments / Resolution	Resolved, this function has been removed.

Issue	Different buckets with the same STBC can be created
Severity	Low
Description	Currently, the createRWABucket function allows to create multiple different buckets but with the same STBC, this might result in multiple different down-stream issues.
Recommendations	Consider marking a STBC as used when it is assigned to a bucket. This includes the unmarking during bucket deletion.
Comments / Resolution	Acknowledged.

Issue	Hardcoded buckets are not created upon deployment
Severity	Low
Description	The architecture incorporates several buckets, a list of all buckets can be found in the Constants.sol contract. The issue is that the createBucket function does not check if an asset in fact is corresponding to a bucket.
Recommendations	Consider modifying the constructor such that all buckets are predetermined.
Comments / Resolution	Acknowledged.

Issue	Usage of safeCoolDownTime might backfire
Severity	Low
Description	The safeCoolDownTime modifier is designed to enforce a cooldown period after the safeAddress is changed. However, this implementation may not provide the intended security benefits and could introduce operational inefficiencies, specifically in an emergency situation where the safe address has been changed less than 24 hours before, no intervention can happen. The primary purpose of a cooldown period is usually to prevent rapid, potentially malicious actions. However, if the safeAddress change itself is initiated by a legitimate action (like a multisig transaction), the cooldown may not add significant security value, since the attacker can directly hack the protocol without changing the safeAddress first.
Recommendations	Consider incorporating such a possible scenario in the thought-process and eventually reevaluate the current implementation.

Comments / Resolution	Resolved.

Issue	Lack of check within addRWATokenToBucket for RWA <-> STBTC compatibility
Severity	Low

Description	The addRWATokenToBucket function allows the DAO address to add any RWA asset to a bucket. However, there is no such check as within the createRWABucket, which ensures that the RWA asset is the collateral for the bucket's stablecoin:
	
	// check if rwa is a collateral of stbc
	if (
	
	ITokenMapping(registryContract.getContract(CONTRACT_TOKEN_MAPPING)).getStbcFromRwa(rwa)
	!= stbc
	) {
	revert InvalidToken();
	}

Recommendations	Consider implementing such a check to prevent down-stream issues.
Comments / Resolution	Resolved.

Issue	deleteRWABucket function is not compatible for buckets with multiple RWAs
Severity	Low
Description	The deleteRWABucket function allows the safe address to withdraw an RWA from a specific bucket and delete the [RWA][BUCKET] storage. The issue with this function is that a bucket can have multiple RWAs, which all share the same distributor, since the distributor is based on the bucket itself. If now a RWA is deleted from a bucket, this also deletes the corresponding distributor: delete _distributor[bucket]; which then means that the distribution for the whole bucket will not work anymore. A bullet was avoided here, since the setBucketDistributor function allows to assign a new distributor to the bucket. However, the whole bucket would be DoS'd until a new distributor is assigned. This is also the reason why this issue is downgraded from high to medium. Moreover, this function seems to not fit in the overall context, as the STBC info is still existent for the corresponding bucket name, this function seems to intend to just remove one single RWA to bucket assignment, which is however already handled within the removeRWATokenFromBucket function.
Recommendations	Consider *not* deleting the bucket's general distributor when removing a RWA from a bucket with multiple RWAs. Moreover, it might make sense to simply removing this function and use the

	removeRWATokenFromBucket function. If this function is desired to delete the whole RWA bucket storage, it must be completely refactored
Comments / Resolution	Resolved.

Issue	createBucket allows for using bucket name which is already assigned to a RWA bucket
Severity	Low
Description	The createBucket function allows the DAO address to create a non-rwa bucket. However, there is no such check which ensures that the bucket parameter is not already assigned to a RWA-bucket. Such an incorrect configuration could result in multiple issues such as overriding the distributor etc. The same issue applies vice-versa on the createRWABucket function as a RWA-bucket can override a non-RWA bucket.
Recommendations	Consider explicitly marking a bucket name as assigned upon both RWA and non-RWA bucket creations and checking if this bucket name is already in use. Alternatively, one can simply check if the corresponding bucket name already has an assigned distributor, this will also prevent the duplicate bucket name usage.
Comments / Resolution	Resolved.

Issue	_safeAddress is private
Severity	Low
Description	Certain variables which might be important for users to inspect should be made public instead of private in an effort to increase transparency. Average users might be unable to directly fetch storage slots using a script.
Recommendations	Consider marking _safeAddress as public.
Comments / Resolution	Resolved.

Issue	Lack of Checks-Effects-Interactions pattern
Severity	Low

Description	Throughout the contract there are one or multiple spots which violate the checks-effects-interactions pattern, to ensure a protection against invalid states, all external calls should strictly be implemented after any checks and effects (state variable changes).
	
	While we acknowledge that certain spots can only be designed in such a way (before-after check), there are spots which can be adjusted properly:
	
	L 620:
	
	IStbc(underlyingToken).mint(address(this), amount);
	// add the new minted token to the amount of stbc backed by this rwa
	rwaInfo.stbcBacked += amount;
	_mintUsDAO(amount * usdaoEmissionRate / SCALAR_TEN_KWEI);

	stbcInfo.profitBalance += amount;
	
	The state variable changes should be applied before the external mint calls.
	
	L 638:
	
	IUsDAO(usDAO).mint(address(this), amount);
	_balance[BUCKET_DAO_USDAO_TREASURY] += amount;
	
	The state variable change should be done before the mint.

Recommendations	Consider following the CEI pattern.
Comments / Resolution	Resolved.

Issue	Balance condition check within mintNewToken is never reached
Severity	Low

Description	The mintNewToken function mints additional STBC to the project whenever the underlying RWA value is larger than the previous cached (stbcBacked).
	
	During this function, there is the following condition check:
	
	L 583:
	
	if (IERC20(rwa).balanceOf(address(this)) < rwaInfo.balance) {
	return currentPrice;
	}
	
	This condition will never be reached as the balance will always

	simultaneously decrease with the contract balance. Even if a RWA donation happens, this will have no impact on this condition check.
Recommendations	Consider simply removing that check.
Comments / Resolution	Resolved.

Issue	Unnecessary return value check for distributor.distribute
Severity	Low
Description	During the distribution of RWA and NON-RWA buckets, the distribute function on the distributor is called and the corresponding return value is fetched. However, after a check within all three distributors we quickly realize that these return values are exactly the same as the provided profitToDistribute parameter.
Recommendations	Consider simply removing the return value caching and use profitToDistribute directly.
Comments / Resolution	Acknowledged. We still recommend fixing this as this would then also allow to implement the checks-effects-interaction pattern.

Issue	burnAsset function contains redundant logic
Severity	Low
Description	Following the minimalistic coding approach, we highly appreciate whenever we identify logic which can be refactored or removed that will result in less possible state transactions. Less code always means less attack surface. The burnAsset function is solely intended to burn LusDAO and mint usDAO. However, it also contains logic for RWAs. This logic is completely redundant and never used.
Recommendations	Consider removing this logic.
Comments / Resolution	Resolved.

Issue	Multicall does not serve any purpose
Severity	Low
Description	Following the minimalistic coding approach, we highly appreciate whenever we identify logic which can be refactored or removed that will result in less possible state transactions. Less code always means less attack surface. The multicall inheritance can therefore be removed, as it is not used anymore once the bucket configurations are simplified.
Recommendations	Consider removing the multicall.
Comments / Resolution	Resolved.

Issue	Encoding for bondDistributorParam is pointless
Severity	Low
Description	The only scenario where the bondDistributorParam is actually used is for the BondDistributor distribution. During the distribution, the param is simply used and the recipient address is fetched. The whole decoding practice is completely pointless as at the end it will always be an address.
Recommendations	Consider removing the decoding practice within the BondDistributor and simply allow for the standard address format to be forwarded as input on the distribute call.
Comments / Resolution	Resolved.

Issue	Lack of validation for setBucketDistributor
Severity	Low
Description	The setBucketDistributor lacks an address(0) validation, this can result in a DoS if set incorrectly.
Recommendations	Consider validating the setBucketDistributor function accordingly.
Comments / Resolution	Resolved.

Issue	Incorrect distributor configuration can result in DoS for distributions
Severity	Informational
Description	The distribute function allows to distribute bucket specific assets to a determined distributor. The most important functions within this flow are the _distributeRegularBucketProfit and _distributeRWABucketProfit functions, as these invoke the distribute function on the specific distributor with the following parameters: - bucket - underlyingToken - profitToDistribute - bondDistributorParam However, it is important to understand that there are three distributors: BondDistributor, LpDistributor and ManualDistributor. Whereas each distributor has its own logic and potentially also hardcoded parameters such as the token and the recipient address. This means that even though the provided parameters are correct, the corresponding distributor can potentially ignore these and use its own storage variables for the distribution logic. This might result in undesired side-effects.
Recommendations	Consider always being very specific and exercise caution whenever a distributor is assigned to a bucket.
Comments / Resolution	Acknowledged.

Issue	Impossible to use same RWA for two different stablecoins
Severity	Informational

Description	As indicated per description, for V1 it is desired to implement multiple stablecoins. However, this will not work as per TokenMapping, each RWA can only have one assigned stablecoin:
	
	function getStbcFromRwa(address rwa) external view returns (address) {
	address stbc = rwaToStbc[rwa];
	if (stbc == address(0)) {
	revert InvalidToken();
	}
	return stbc;
	}
	
	This function is then used for minting purposes to fetch the corresponding stablecoin from a RWA, which means that only one possibility will ever exist.
	
	*This issue is also added as a global issue.

Recommendations	Consider refactoring the codebase for V1 in an effort to support multiple STBCs, multiple spots in the codebase would need to be adjusted.
Comments / Resolution	Resolved, this is not intended.

Issue	Contract will never be in a state where sbtc.profitBalance is > 0
Severity	Informational
Description	Whenever the mintNewTokens function is invoked, any potential value increase of the underlying RWAs is denominated in STBC and minted to the protocol as profit. Subsequently, the distribute function is called with the profitToDistribute parameter = 0. This will result in the full profitBalance being distributed. The contract will therefore never be in the state where sbtc.profitBalance > 0. Therefore, any logic within the contract that attempts to transfer profitBalance out is redundant.
Recommendations	Consider if it might make sense to refactor the contract and remove all redundant functions.
Comments / Resolution	Not resolved, the contract still incorporates functionality which attempts to withdraw any non-zero profitBalance. However, since the only function which increases the profitBalance is the _mintNewToken function which inherently invokes the distribute function with amount = 0, any increase in profitBalance will be vanished in the same transaction.

Issue	Typographical issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below: L 171:

	event AssetBurnt(bytes32 indexed bucket, address indexed account, uint256 amount); The account parameter is incorrect, it should be token. L 702: IStbc(asset).burn(amount); This burns LusDAO, not STBC.
Recommendations	Consider fixing the aforementioned typographical issues.
Comments / Resolution	Acknowledged.

BondDistributor

The BondDistributor is a simple helper contract which exposes the distribution of usdDao from the specific buckets to the receiver, which must be a valid bond (LsausUSD).

It is solely invoked by the BucketDistribution contract and should only be set as a distributor for any usDAO related buckets, respectively BUCKET_DAO_LOCKED_SAVINGS_ACCOUNT. The corresponding usDAO bucket is a non-rwa bucket.

Threat Modeling:

Since this contract is responsible for interaction with the LsausUSD contract and supplying the correct amount of tokens upon yield generation, it is important to ensure that sufficient tokens are existent in this contract. Corresponding issues are aggregated in the LsausUSD contract.

Issue	Decoding practice is pointless
Severity	Low
Description	As already mentioned within the BucketDistribution contract, the provided bytes parameter is pointless and a simple address parameter should be provided. Therefore, the decoding practice can be removed.
Recommendations	Consider removing the decoding practice. This must also be adjusted within the LsausUSD contract, the encoding should be removed and the simple “address(this)” parameter should be forwarded on the distribute call.
Comments / Resolution	Resolved.

Issue	IERC20 important is unused
Severity	Informational
Description	The IERC20 interface is imported but never used, in fact the contract directly uses the imported usDAO contract for casting, which works as well.
Recommendations	Consider removing the IERC20 import and assignment to safeERC20.
Comments / Resolution	Resolved.

LpDistributor

The LpDistributor contract is a simple distributor contract which transfers usDAO from the BucketDistribution contract in, then locks the usDAO to receive lusDAO and then deposits the lusDAO in a specific predetermined receiverPool.

At this point, we do not have any knowledge about the receiverPool and any potential corresponding issues from that interaction. The function selector indicates that this is a reward token deposit. One must be cautious that this does not manipulate the reward rate, otherwise users could frontrun the distribution with a large deposit and then distribute the rewards to the receiverPool.

This audit does not cover potential integration issues with the receiverPool.

It is important to ensure that this distributor is only assigned to usDAO buckets, which is a non-rwa bucket.

Threat Modeling:

The main issues around this contract lie within the correct integration by the BucketDistribution contract and the correctness of the hard coded parameters. Moreover, the integration of the receiverPool must be correct.

Issue	Typographical Issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below: All comments indicate that this contract distributes profit to a bond, however, that is not the case: L 21: /// @title BucketDistributor - Manages profit distribution to the bond buckets. /// @notice This contract handles the distribution of profits to bond

	buckets and allows the receiver address to be updated by authorized users. /// @author Usual Tech Team L 77: /// @notice Distribute the profit to the bond /// @param _bucket the bucket to distribute /// @param _token the token to distribute /// @param profitBalance the profit balance to distribute /// @param bondDistributorParams the data to decode /// @return profitDistributed the profit actually distributed
Recommendations	Consider fixing the aforementioned typographical issues.
Comments / Resolution	Acknowledged.

ManualDistributor

The ManualDistributor contract is a simple distributor contract which distributes a specific token from the BucketDistribution contract to a predetermined receiver contract. Contrary to the BondDistributor and the LpDistributor, this contract is specifically designed without a hardcoded token, therefore it is likely used for the distribution of USDT, USDC, LsausUSD and STBC, which means it can either be used for non-rwa buckets (USDT, USDC, LsausUSD) or for RWA buckets (STBC).

The correct configuration (assignment to a bucket) relies on the usualTech development team.

Threat Modeling:

The main issues around this contract lie within the correct integration by the BucketDistribution contract and the correctness of the hard coded parameters. Since this is a simple distributor contract and the access control is validated, the attack surface is limited.

Issue	Typographical Issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below: All comments indicate that this contract distributes profit to a bond, however, that is not the case: L 16: /// @notice Distribute the profit to the bond L 74: /// @notice Distribute profits to the bond bucket.
Recommendations	Consider fixing the aforementioned typographical issues.
Comments / Resolution	Acknowledged.

Factory

LsausUSDFactory

The LsausUSDFactory is a simple factory contract that allows the DAO address to deploy LsausUSD tokens. Once deployed, each token is added as a bond. Additionally, the DAO address can mark a bond as removed/not added. A LsausUSD bond can be created with the following data:

- name: ERC20 name

- symbol: ERC20 symbol

- bondStartDate: Start of the bonding period, used to determine passed days and total duration

- duration: Added towards the bondStartDate, determines the end of the bondind period when users can withdraw their STBC

- saversPercentage: Yield in BPS over the full bonding period

- transferabilityFee: Tax when transferring LsausUSD

- stbcToken: Token which can be used to obtain bond

- bucket: Bucket which receives the transfer-tax

It is important to mention that bonds with the same name can be active and deployed simultaneously.

Threat Modeling:

The attack surface for this contract is limited since all function entries are restricted. Issues can occur if there is a DoS of the deployment or due to misconfiguration of the isBond mapping as well as misconfiguration of deployment parameters.

Issue	Removal of bond is irreversible and can result in permanently locked STBC
Severity	**High**
Description	The DAO address can remove bonds from the contract's very own registry via the removeBond function. This will have an inherent impact on the distribution logic as any removed bond will essentially be broken due to the revert in the distribution call. If a bond is ever accidentally removed, its functionality can never be restored, resulting in user funds being irreversibly locked in the bond due to a revert within the unwrap function.
Recommendations	Consider implementing an emergency function which allows to re-add a bond.
Comments / Resolution	Resolved, the isBond call is not used anymore. Instead the distribute flow will transfer the usDAO to the factory contract.

Issue	Lack of validation for deployment parameters
Severity	**Low**
Description	The createLsausUSD function lacks a proper parameter validation, insufficient validation can result in exploits or undesired scenarios when this function is invoked. a) The bondStartDate should be in the future (is checked within

	the actual constructor but should revert early) b) The duration should be a reasonable time c) The saversPercentage should be a reasonable number The transferabilityFee should not be larger than 5-10%
Recommendations	Consider implementing these essential validations.
Comments / Resolution	Partially resolved, most validations are implemented.

Issue	**Different variables can be made immutable**
Severity	Informational
Description	Variables which are defined within the constructor and never changed afterwards can be marked as immutable, this will include these variables directly into the contracts bytecode, which will save gas whenever these variables are accessed. L 25: IRegistryContract private _registryContract; L 26: IRegistryAccess internal _registryAccess;
Recommendations	Consider marking the aforementioned variables as immutable.
Comments / Resolution	Resolved.

SbtcFactory.sol

The SbtcFactory is a simple factory contract that allows the DAO address to deploy STBC tokens. Once deployed, each token is added as a valid STBC to the contract's internal registry. This information is then used via the isStbc view-function throughout the protocol.

It is important to mention that STBCs with the same name can be active and deployed simultaneously.

Threat Modeling:

Due the access control mechanism the attack surface for this contract is limited. Issues could arise due to the accidental removal of STBC from the contract's registry.

Issue	Removal of STBC is irreversible and can result in revert of setStbcToRwa function
Severity	Medium
Description	The DAO address can call the removeStbc function in an effort to remove the STBC from the contract's internal registry. This information is used within the TokenMapping contract within the setStbcToRwa and setRwaToStbc functions. Specifically the state which is set within the latter function has an impact on standard minting, minting via the MintEngine and redeeming and can result in a DoS. Contrary to the issue within the LsausFactory, this issue has only rated as medium for the following reasons: a) DoS would only happen in two-step scenario: 1. removing from the registry 2. call of removeRwaToStbc function. b) Backed RWAs could still be emergency withdrawn.

Recommendations	Consider implementing an emergency function which allows to re-add a STBC.
Comments / Resolution	Resolved, this contract has been removed completely.

Issue	Potentially incorrect reset mechanism for _stbcIdToAddress mapping
Severity	Low
Description	Whenever the _removeStbc function is invoked, the following condition is checked: if (_stbcIdToAddress[_stbcLastId] == stbc) { which will then clear the corresponding mapping. However, this condition is only true if the to be removed STBC is in fact the latest deployed one. If there is another one deployed in between, this condition will not be true, which results in the mapping not being cleared.
Recommendations	Consider checking if this is desired, if not consider clearing the mapping under any circumstance.
Comments / Resolution	Resolved, this contract has been removed completely.

Issue	Unused variable(s) unnecessarily increase code size
Severity	Informational
Description	Variables which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers. L 27: IStbcFactory internal _stbcFactory;
Recommendations	Consider removing the aforementioned unused variable.
Comments / Resolution	Resolved, this contract has been removed completely.

Issue	Incorrect revert reason for removeStbc
Severity	Informational
Description	Within the removeStbc function, the function will revert if the STBC is not part of the internal registry. However, the revert reason is the following: revert NullAddress(); which is not necessarily correct.
Recommendations	Consider implementing the correct revert reason.
Comments / Resolution	Resolved, this contract has been removed completely.

MintEngine

MintEngineOrder

The MintEngineOrder contract is the entry contract for users to create two types of offers: RWA offer and Minter offer. Moreover, governance ca whitelist minter and cp (RWA) tokens.

Offers explained in detail:

1. OfferMinter: A user can offer a specific amount of minter tokens, which is currently limited to USDT and USDC . The offerer expects a STBC in the equivalent value as the provided USDT or USDC

2. OfferCpToken: A user can offer a specific amount of CP (Collateral Provider) token, which is currently limited to USYC, a US Yield Coin (https://usyc.docs.hashnote.com/getting-started/introduction), moreover, a preferred token (USDT or USDC) is provided. The offerer expects to receive either USDT or USDC.

It is important to mention that while the protocol indicates that solely these tokens will be used for now, it is not excluded that any other tokens will be implemented in the future. Governance can simply add and remove tokens from the minter and Cp whitelist.

Whenever an order is created, the token amount is being normalized to 18 decimals which simplifies the following operations.

The different order flows are illustrated as follows:

createOfferMinter:

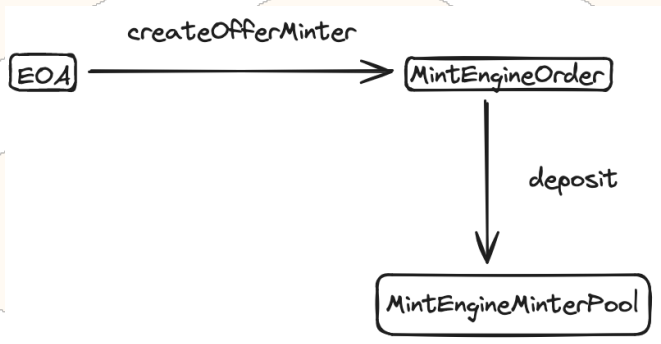
Scenario 1: Non-Existing RWA offers:

1. Alice creates a minter offer with USDT

2. The RWA orderbook is empty: deposit the USDT amount into the MinterPool

3. Alice can now cancel the order or wait for incoming RWA orders.

4. Once a RWA offer was created and matched with Alice's minter offer, Alice can now claim her stbc, either full or partially, depending on the RWA offer size and the progress of the current round (currentId).



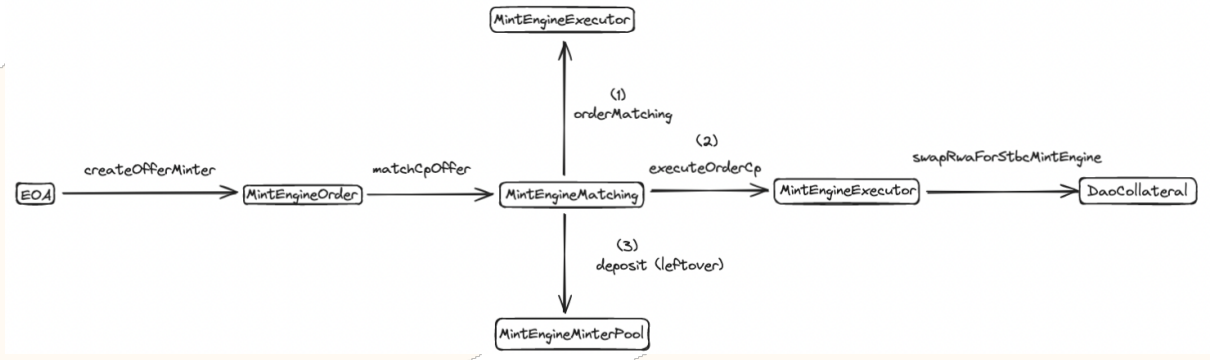
Scenario 2: Existing RWA offers:

1. Alice creates a minter offer with USDT

2. An automatic attempt is executed to match the offer with any existing RWA offers for USDT in the orderbook.

3. If offers match or partially match, the order is executed, which invokes the swapRWAForSTBCMintEngine function, resulting in USDT being transferred to the RWA-provider, SBTC minted to the USDT provider and the RWA being transferred to the bucket. For this call, the swapRwaForStbcMintEngine function within the DaoCollateral contract is being leveraged.

4. If there is a leftover amount, which means the offer was only partially filled, this is deposited into the MinterPool, waiting to be filled by newly provided RWA-Offers. It is important to mention that the cancellation (withdrawal) of pooled tokens can only happen if there are virtual orders in the minter pool for the corresponding deposit ID.



createOfferCpToken (RWA Offer):

Existing Minter offers:

1. Alice creates an RWA offer.

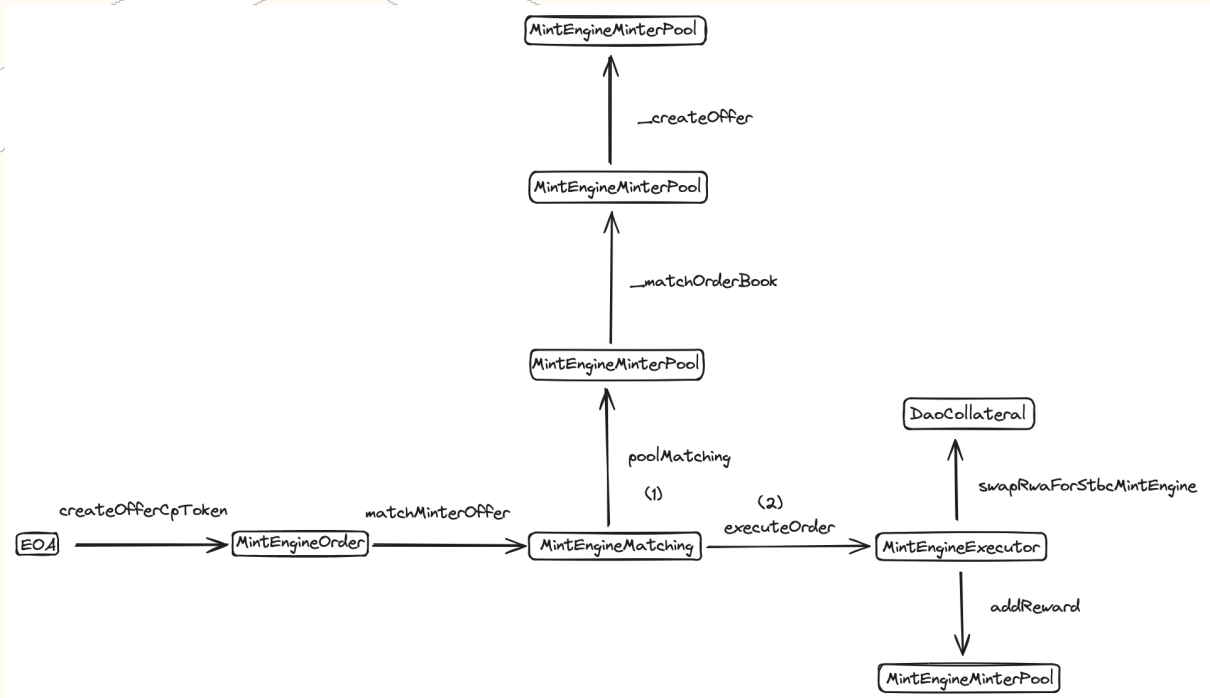
2. This offer is now matched with the MinterPool offers. A few different scenarios can happen here, depending on the state of the MinterPool and the RWA offer size:

Scenario 1: The RWA offer is the first offer and is covered by the amount in the minter pool:

3. A virtual offer with the amount of minter tokens in the MinterPool contract is created.

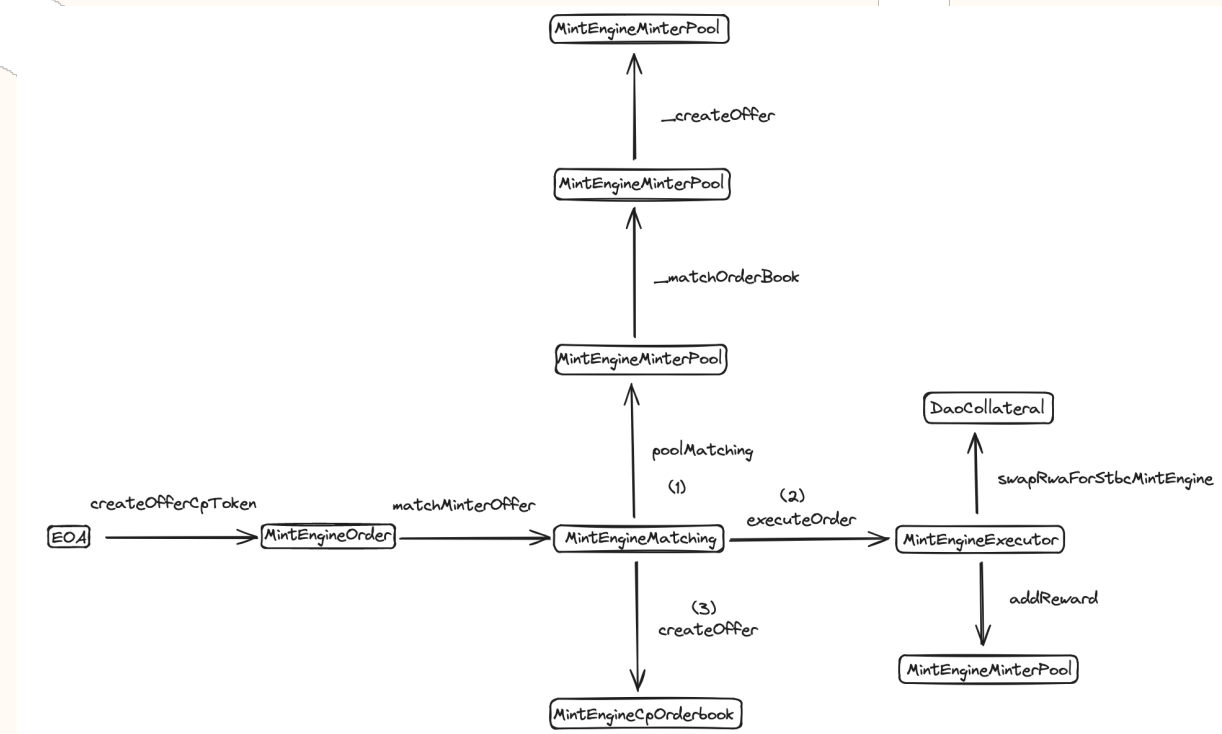
4. The RWA offer is now matched with the created virtual offer and is fully covered by the virtual offer.

5. The order is executed which locks the RWA into the bucket, transfers USDT from the minter pool to the RWA provider and assigns the reward for the USDT provider to claim it. A potential leftover from the virtual offer can exist, which will be matched with subsequent RWA offers.



Scenario 2: The RWA offer is the first offer and is not covered by the amount in the minter pool:

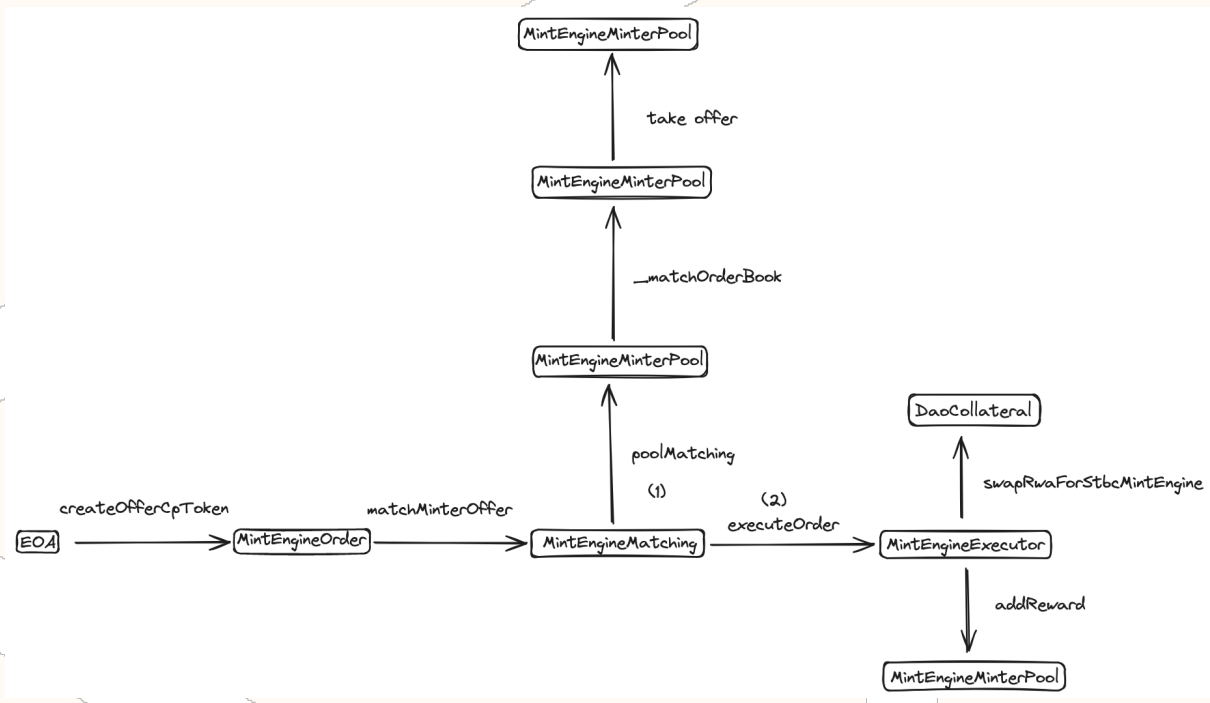
Exactly the same scheme as with the aforementioned flow, additionally an order in the OrderBook is created with the uncovered amount.



Scenario 2: There is already an existing virtual offer and the provided RWA offer is covered by it:

3. The RWA offer is now matched with the created virtual offer and is fully covered by the virtual offer, the virtual offer is decreased by the value of the RWA offer.

4. The order is executed which locks the RWA into the bucket, transfers USDT from the minter pool to the RWA provider and assigns the reward for the USDT provider to claim it.



Scenario 4: There is already an existing virtual offer and the provided RWA offer is not covered by it. The pooled minter funds (includes deposits for the subsequent ID) cover the RWA offer:

3. The RWA offer is now matched with the created virtual offer and is not fully covered by the virtual offer.

4. Since the pooled minter value can cover the leftover from the RWA offer, a new offer for the next id is created and an additional transferInfo with the remaining value is created.

5. A bug was found during that scenario, it needs to be refactored by the development team.

Scenario 5: There is already an existing virtual offer and the provided RWA offer is not covered by it. Funds from the subsequent ID are existent but will not fully cover the rollover amount from the RWA offer.

3. The RWA offer is now matched with the created virtual offer and is not fully covered by the virtual offer.

4. The pooled minter value from the subsequent deposit cannot cover the rollover from the RWA offer.

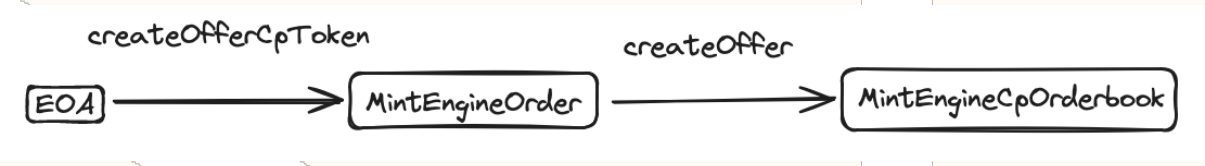
6. A new offer is created with the pooled funds and an additional transferInfo is created with the pooled value.

7. A bug was found during that scenario, it needs to be refactored by the development team.

Non-Existing Minter offers:

1. Alice creates an RWA offer, there are no existing minter offers.

2. Alice's RWA offer is pushed into the OrderBook, waiting to be picked up by incoming minter offers.



Threat Modeling:

Since the MintEngineOrder serves as an entry point for users to interact with the complex MintEngine module, focus lies mostly on input validation and the correctness of configurational settings.

Issue	Governance Privilege: Whitelisting and removing of assets
Severity	Governance
Description	The governance privilege of whitelisting and de-whitelisting of assets can have a huge impact. It is theoretically possible for a malicious governance body to whitelist any asset as RWA asset and drain all minter offers.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue	Lack of USD conversion for RWA asset can result in multiple issues for the orderbook flow
Severity	High
Description	Whenever a RWA offer is created, there are two possibilities: a) Push the RWA offer in the OrderBook b) Match the RWA offer with funds in the MinterPool For both flows, one RWA asset (1e18) is always treated as 1 USDT/USDC. This assumption is totally incorrect as a) a depeg can happen and b) there are plenty of RWA assets which are not worth 1 USD If we now assume the following scenario in V0: 1. Alice creates offer with 100 RWA (100 USD) 2. RWA price depegs to 0.95, offer value is now 95 USD

	3. Bob wants to provide USDT for STBC, creates an offer of 100 USDT 4. Offer is matched with Alice's offer, Alice will receive 100 USDT. However, Bob will only receive 95 STBC for 100 USDT.
Recommendations	A fix for that issue will require refactoring of the logic, ideally the value of the provided RWA is converted into USD and then the provided USD value is used for accounting and matching purposes. However, this will then result in potential issues whenever the offer sits in the orderbook, as the USD value will fluctuate in the meantime. Alternatively, a dynamic approach can be used which converts the provided amount during each process to USD and then does the accounting based on the USD value. This might be the most ideal solution but requires advanced development effort to ensure this cannot get exploited and will not result in any arithmetic operations to revert. However, as one can see, potential solutions always come with a compromise. A temporary solution for V0 would be a depeg check during the offer creation and a revert if the value fluctuates more than a few BPS. However, also this will result in issues if the value fluctuates for existing offers.
Comments / Resolution	Partially resolved, a dynamic adjustment approach was made. For instance, if a user provides a RWA asset, this is pushed into the orderbook with the nominal rwa value. Then upon fetching, it is calculated how much RWA matches the provided USD value. Contrary, upon the minter offer creation, the nominal USDT amount is pushed into the MinterPool and upon RWA offer creation, the USD value from the RWA amount is used to match with the existing USDT

	offers. Unfortunately, there were some issues introduced in this path which we raised in the corresponding section. After these issues are fixed, the mechanism must be re-assessed.

Issue	Assumption of USDT/USDC being naturally worth 1 USD can result in loss of funds for RWA provider
Severity	High
Description	Similar to the aforementioned issue but with slightly less impact can this issue be understood. Due a potential price fluctuation multiple issues can occur. Example: 1. Alice deposits 100 USDT 2. USDT depegs to 0.95 or is already 0.95 during the offer creation, there is no difference between both scenarios for the context. 3. Bob creates offer with 100 RWA (100 USD) 4. Alice receives 100 STBC 5. Bob receives 100 USDT, only worth 95 USD.
Recommendations	Consider following a similar approach as in the aforementioned issue.
Comments / Resolution	Partially resolved, see issue before.

Issue	Duplicate deposit within createMinterOffer allows for stealing tokens from the minter pool
Severity	**High**

Description	The createOfferminter function allows anyone to steal tokens from the MinterPool. First of all, it is important to understand which specific scenarios of a minter offer creation exist:
	
	Scenario 1:
	
	No RWA offers are existing in the MintEngineCpOrderBook contract, the offer is forwarded to the minter pool, which is then treated on a FiFo approach upon RWA offer creations.
	
	Scenario 2:
	
	RWA offers are existing, the provided minter offer is matched with the existing RWA offers, if the RWA offers are insufficient to match the minter offer, the leftover is deposited to the minter pool.
	
	Now let's come to the root-cause: The deposit to the minter pool is happening twice:
	
	IERC20(token).safeTransferFrom(msg.sender, _executor, amount);
	amount = _scaleAmountByDecimals(token, amount);
	_mintEngineMinterPool.deposit(msg.sender, token, amount);
	if (_mintEngineCpOrderbook.isCpOrderbookEmpty(token)) {
	return;
	}
	_mintEngineMatching.matchCpOffer(msg.sender, token, amount);
	}
	
	PoC of the exploit:
	
	1. Bob creates an RWA offer with a value of 100 USD, this offer is

	pushed into the order book. At this point there is no existing minter offer.
	
	2. Alice creates a minterOffer with 100 USDT, a deposit is directly happening to the minter pool, which increases the following storage variables:
	
	_user[Alice].balancePerOffer[USDT][1] = 100;
	_token[USDT].balancePerOffer[1] = 100;
	
	However, it will also match with the existing RWA offer, which then automatically transfers the STBC to Alice and the USDT to Bob, which means that Alice's allocation is completely consumed.
	
	3. Alice can now simply call claim and withdraw 100 USDT, while she in fact already received 100 STBC(presumed another user deposits afterwards).
	
	This PoC can be reproduced with several scenarios.
	
	The problem lies within the fact that the deposit call to the MinterPool is executed not only during a phase of empty RWA offers but also if there are existing RWA offers.

Recommendations	Consider removing the duplicate call. It is important to mention that a full re-audit is necessary for the MintEngine module.
Comments / Resolution	Partially fixed. The duplicate deposit was removed, however, in the scenario of a match, the user will still have the deposit in the MinterPool. This will break the accounting.

Issue	No distinction between different RWA offers can result in incorrect matching execution
Severity	Medium
Description	Even though V0 only intends to have one RWA (USYC), the possibility exists that multiple RWAs can be whitelisted and therefore used to create offers. The problem lies within the fact that there is absolutely no distinction for different RWAs, they will be just pushed in the exact same order book (depending on the preferred asset) and whenever a minter offer is created, the next offer in the order book will be fetched, no matter which RWA this includes. Vice-versa this issue also applies for existing minter offers whenever a RWA offer is created, the minter offerer cannot determine which RWA to accept. *This issue is only rated as medium severity due to the fact that there will only be one existing RWA for V0.
Recommendations	For V0, this issue can be either acknowledged or the contract can be rewritten in such a manner that only one RWA can be whitelisted. For V1, the whole MintEngine needs to be refactored completely to allow minter offerers to determine for which RWA their offer should apply.
Comments / Resolution	Acknowledged, this will be tackled in V1.

Issue	Lack of kyc check during createOfferCpToken can result in forced cancellation and partial loss of funds
Severity	Medium
Description	The createOfferCpToken allows any user to provide a RWA asset in an exchange for a minter asset (USDT/USDC). The issue within this flow is that even non-kyced users can create an offer, however, this offer can then never be executed due to the following check within the swapRwaForStbcMintEngine function: uint256 ubo = _registryContract.getUBO(rwaProvider); which will simply revert if a user is not kyc-ed. This will then result in admin intervention where the order needs to be force closed, resulting in a loss for the user.
Recommendations	Consider simply executing a kyc check upon beginning of the createOfferCpToken function.
Comments / Resolution	Not resolved, a non-kyc-ed user can still place a RWA offer which is then pushed inside the MintEngineCpOrderBook. Once this offer is then fetched, it will revert because the RWA provider does not have an assigned UBO. This will brick the MintEngine mechanism until governance manually cancels this offer.

Issue	Decimal normalization is flawed due to division before multiplication
Severity	Medium
Description	Users can create offers via the entry functions createOfferMinter and createOfferCpToken. These functions will scale the provided RWA and USDT/USDC amounts to 18 decimals for later matching purposes. The problem here is that the scaling only works to the upside and not to the downside. Tokens with >18 decimals can therefore never be used as it will round down to zero due to a division before multiplication. Interestingly, in the other contracts the normalization is done correctly, without the brackets. *This issue is only rated as medium severity since in V0 such tokens will not be used. However, this should be fixed under all circumstances.
Recommendations	Consider simply following the same normalization as throughout the rest of the protocol.
Comments / Resolution	Resolved, this has been removed.

Issue	Lack of validation for zero amounts within createOfferMinter
Severity	Low
Description	> The createOfferMinter function allows users to create USDT/USDC offers. This function does not check for amount = 0 as input. This can result in undesired side-effects.
Recommendations	Consider validating the function parameter accordingly.

Comments / Resolution	Acknowledged.

Issue	Hardcoded MINIMUM_ORDER_RWA might backfire in future
Severity	Low
Description	Currently, users can only create an RWA offer with >= 10_000 RWA tokens. This might backfire in the future whenever a different configuration is desired. Not to mention if there will be other RWAs added which are worth more than 1 USD per asset.
Recommendations	Consider making this parameter changeable.
Comments / Resolution	Resolved.

Issue	Private variables
Severity	Low
Description	Certain variables which might be important for users to inspect should be made public instead of private in an effort to increase transparency. Average users might be unable to directly fetch storage slots using a script. L 39: address private immutable _executor; L 47 FF: IRegistryContract private _registryContract;

	IRegistryAccess private _registryAccess; IMintEngineMatching private _mintEngineMatching; IMintEngineCpOrderbook private _mintEngineCpOrderbook; IMintEngineMinterPool private _mintEngineMinterPool;
Recommendations	Consider marking the aforementioned variables as public.
Comments / Resolution	Resolved.

Issue	Typographical Issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below: L 20: CONTRACT_RWA_FACTORY, This import is unused. L 41 FF: uint256 public currentIdMinter; uint256 public currentIdCp; uint256 public lastTookMinter; uint256 public lastTookCp; These variables are unused.
Recommendations	Consider fixing the aforementioned issues.

Comments / Resolution	Acknowledged.

Issue	Certain variables can be made immutable
Severity	Informational
Description	Variables which are defined within the constructor and never changed afterwards can be marked as immutable, this will include these variables directly into the contracts bytecode, which will save gas whenever these variables are accessed. L 47 FF: IRegistryContract private _registryContract; IRegistryAccess private _registryAccess; IMintEngineMatching private _mintEngineMatching; IMintEngineCpOrderbook private _mintEngineCpOrderbook; IMintEngineMinterPool private _mintEngineMinterPool;
Recommendations	
Comments / Resolution	Resolved.

Issue	Unused events
Severity	Informational

Description	Certain events are unused, this does not only confuse third party reviewers but will also affect other parties which rely on the correctness of event emission.
	
	/// @notice Event emitted when tokens are deposited for minting.
	event DepositMinter(address indexed token, uint256 amount);
	
	/// @notice Event emitted when tokens are deposited for cross-pool (CP) transactions.
	event DepositCp(address indexed token, uint256 amount);
	
	/// @notice Event emitted when a token is added to the whitelist.
	event AddWhitelistToken(address indexed token);

Recommendations	Consider removing or using these events.
Comments / Resolution	Resolved.

MintEngineCpOrderBook

The MintEngineCpOrderBook contract is a simple orderbook implementation for RWA offers. Once a RWA offer is created, it is first checked if the MinterPool has any funds to cover this order. Now there are three scenarios in total:

1. MinterPool has sufficient funds to cover the RWA offer: Orderbook mechanism is not invoked.

2. MinterPool has insufficient funds to cover the RWA offer completely but can partially cover it: The offer is matched partially and the leftover amount is used for an order creation in the orderbook.

3. MinterPool has no funds: The full RWA offer is created as order in the orderbook.

Now, as with all orderbooks, once an order is created it will simply sit in the books until it is invoked. This happens upon a minter offer creation on a FiFo basis. Once a minter offer is created, it will check if there are any outstanding RWA offers and will then match these RWA offers with the minter offer using the orderMatching mechanism. This is a simple loop, executed over all RWA offers until the minter offer is covered fully or until all RWA offers have been depleted.

Users with outstanding RWA offers can cancel these offers for a 0.005% fee, which is directly forwarded to the bucket.

Threat Modeling:

The main attack surface for this function is the cancellation logic and the order matching algorithm.

Issue	Incorrect orderID assignment can be abused by malicious user to steal all RWAs in the queue
Severity	High

Description	Whenever a user creates an RWA offer which is meant to be pushed in the OrderBook, the currentId is increased before the offer creation:
	
	++currentId[preferredToken]; // We do ++ so we can use 0 for not found offers
	
	this will result in the offerId starting for the first order with 1 instead of zero. However, during the creation of the offer in the Queue library, the offer will be naturally pushed to the first index, which is zero:
	
	function create(List storage queue, Offer memory offer) internal {
	uint128 backIndex = queue._end;
	queue._value[backIndex] = offer;
	unchecked {
	queue._end = backIndex + 1;
	}
	emit OfferCreated(offer);
	}
	
	Upon cancellation, the correct order is fetched:
	
	Queue.Offer memory offer = _offer[preferredToken].getOfferFromId(offerId);
	
	this function simply loops over the queue until the correct offerId is fetched.
	
	However, the offerId is provided for the cancel call in the Queue

library, which is 1 in that scenario:

_offer[preferredToken].cancel(offerId);

taking a look at the cancel function, we will quickly realize that the parameter is expected to be the index, not the offerId:

```
        function cancel(List storage queue, uint128 index) internal
returns (Offer memory) {
        if (isEmpty(queue)) revert Empty();
        Offer memory value = queue._value[index];
        value.status = OfferStatus.CANCELED;
        queue._value[index] = value;
        emit OfferCancelled(value);
        return value;
        }
```

This flaw can be abused by a sophisticated attacker to drain all RWAs in the Queue.

Poc:

1. Alice creates a RWA offer for USDT with 10_000

Offer from Alice:
offerId = 1
token = USDT
amount = 10_000
timestamp = current TS
owner = Alice
status = WAITING

this offer struct is now pushed to queue._end:

```
        function create(List storage queue, Offer memory offer)
internal {
        uint128 backIndex = queue._end;
        queue._value[backIndex] = offer;
        unchecked {
```

```
            queue._end = backIndex + 1;
            }
            emit OfferCreated(offer);
            }
```

which is in that scenario the first index, hence 0
-> queue._value[0] = offerAlice;

2. Bob and Charles also create RWA offers:

Offer from Bob:
offerId = 2
token = USDT
amount = 10_000
timestamp = current TS
owner = Bob
status = WAITING
-> queue._value[1] = offerBob;

Offer from Charles:
offerId = 3
token = USDT
amount = 10_000
timestamp = current TS
owner = Charles
status = WAITING
-> queue._value[2] = offerCharles;

3. Alice calls cancelOffer with her offer (offerId = 1)
-> it correctly fetches Alice's offer from the ID = 1, which will fetch the offer from index 0.
-> it will invoke cancel for Alice's offerId (ID = 1), this will set the offer to CANCELED. However, it will not set Alice's offer to CANCELED but it will do that with Bob's offer, remember, the provided index = 1:

```
            function cancel(List storage queue, uint128 index) internal
returns (Offer memory) {
            if (isEmpty(queue)) revert Empty();
            Offer memory value = queue._value[index];
value.status = OfferStatus.CANCELED;
```

	queue._value[index] = value;
	emit OfferCancelled(value);
	return value;
	}
	
	4. The function will continue as expected with Alice receiving funds while Bob's offer was cancelled.
	
	5. Alice can repeat this exploit, stealing *all* RWA in the queue.
	
	Interestingly, this issue does not have an impact if no cancellation happens, since the take function will naturally just take the next offer from the queue.
	
	This flaw can result in multiple other dramatic results.

Recommendations	Consider simply using an orderId which matches the corresponding _begin index.
Comments / Resolution	Resolved, the orderId is now always matching the corresponding _end within the Queue.sol contract.

Issue	Lack of access control for orderMatching allows malicious user to empty all orders, resulting permanently stuck funds
Severity	**High**
Description	The orderMatching function is responsible for matching incoming minter offers with existing RWA offers. This function is solely meant to be invoked via the MintEngineMatching.matchCpOffer function. However, this is not the case as this function is publicly callable which allows anyone to just empty the full order book, resulting in permanently stuck funds
Recommendations	Consider implementing a proper access control for this function, ensuring only the MintEngineMatching contract can invoke it.
Comments / Resolution	Resolved, the orderMatching mechanism is now only callable by the MintEngineMatching contract.

Issue	Logical error within cancelOffer will result in a loss of most fees
Severity	**High**
Description	Upon offer cancellation, a 0.005% fee is taken from the RWA amount which is then transferred to the bucket: _daoCollateral.swapRwaForStbcMintEngine(address(bucket), address(bucket), token, fees); The problem here lies within the fact that it is not done via a simple transfer and add to the bucket but rather using the swap and mint mechanism. This will result in several issues: 1) Since this minting mechanism mints a fee to the rwaProvider and stableProvider, this means a fee will be minted directly to the bucket:

	uint256 lusdaoInstant = _distributeUsDAO(amountUsDAOToMint, priceInUSD, rwaProvider, stableOwner); However, this fee is not added to the bucket, which means it is just permanently locked. 2) Since a swap and mint is executed the bucket will receive the STBC and the RWA: IStbc(tokenToMint).mint(stableOwner, priceInUSD); However, the STBC is not incorporated in the bucket profit. From our opinion, the swap and mint mechanism here should be completely avoided, as it does not really make any sense. Rather the RWA fee should be transferred directly to the bucket and added towards it. Moreover, this will artificially inflate the STBC circulation, resulting in a lower CBRCoef.
Recommendations	Consider simply transferring the RWA to the bucket and adding it. However, when implementing this change, potential side-effects must be incorporated in the thought and testing process. A full re-audit of the MintEngine module is necessary anyways.
Comments / Resolution	Not resolved, the STBC as well as LusDAO tokens are directly minted to the BucketDistribution contract but never accounted for. This can be simplified by just sending the fees straight to the BucketDistribution contract and add it to the corresponding bucket.

Issue	Anti-DoS implementation for orderMatching logic is flawed
Severity	Medium
Description	The orderMatching function is invoked upon the matchCpOffer function during the createOfferMinter call. The basic idea behind this logic is to fetch all existing RWA offers until the desired amount is reached. This function incorporates a DoS check by ensuring that the numOrderTook variables is below 10. However, this variable is never incremented, rendering this check effectively useless.
Recommendations	Consider properly incrementing this variable.
Comments / Resolution	Resolved.

Issue	CancelOrder will revert if BucketDistribution contract is not KYC-ed
Severity	Medium
Description	The cancelOrder function allows a user to cancel their order, a fee on the RWA is taken and forwarded to the DaoCollateral contract via the swapRwaForStbcMintEngine function. The issue is that the rwaProvider in this scenario is the bucket contract, which is obviously no natural person and therefore not kyc-ed. However, in an effort to not revert, this address must be kyc-ed (have a corresponding UBO identifier).
Recommendations	Consider adding a UBO identifier to the BucketDistribution address.
Comments / Resolution	Acknowledged, this will be done manually.

Issue	Multicall does not serve any purpose
Severity	Low
Description	Following the minimalistic coding approach, we highly appreciate whenever we identify logic which can be refactored or removed that will result in less possible state transactions. Less code always means less attack surface. The multicall inheritance can therefore be removed, as it is not used anymore once the bucket configurations are simplified.
Recommendations	Consider removing the multicall.
Comments / Resolution	Acknowledged, potential side-effects can occur from this.

Issue	Typographical Issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below: L 5: import {ERC20} from "openzeppelin-contracts/token/ERC20/ERC20.sol"; Unused import. L 8: import {SafeERC20} from "openzeppelin-

contracts/token/ERC20/utils/SafeERC20.sol"

Duplicate import.

L 18:

import {IOracle} from "src/interfaces/oracles/IOracle.sol";

Unused import.

L 29:

CONTRACT_MINT_ENGINE_MATCHING,

Unused import.

L 31:

CONTRACT_MINT_ENGINE_ORDER,

Unused import.

L 33:

CONTRACT_ORACLE_USUAL

Unused import.

..... up to

L 37:

Unused import.

	L 44 - 48: Unused imports. L 66-81 Unused variables/declarations/modifiers L 100: Unused modifier. L 108: Unused modifier. L 116: Unused modifier. L 281: Unused function. L 308: Unused function.
Recommendations	Consider fixing the aforementioned typographical issues.
Comments / Resolution	Resolved.

MintEngineMatching

The MintEngineMatching contract is an intermediary contract which is invoked upon minter and RWA offer creations. It is invoked upon the creation of these offers and handles the matching as follows:

a) RWA offer creation: Forwards the call to the MinterPool contract which is undergoing the pool matching mechanism. Afterwards it will execute the order and eventually create a leftover RWA offer in the orderbook, if the MinterPool had insufficient funds for covering the initial order.

b) Minter offer creation: Forwards the call to the RWA orderbook which then matches outstanding RWA offers and executes the order. Afterwards it will create a leftover minter offer if the RWA offers were insufficient to cover the minter offer. This is just a simple deposit in the MinterPool contract which is treated on a FiFo basis.

Threat Modeling:

The main risk for this contract is the interconnection with the rest of the MintEngine architecture.

Issue	Certain variables can be made immutable
Severity	Informational
Description	Variables which are defined within the constructor and never changed afterwards can be marked as immutable, this will include these variables directly into the contracts bytecode, which will save gas whenever these variables are accessed. L 40: IRegistryContract public registryContract; L 41: IRegistryAccess public registryAccess;
Recommendations	Consider marking these variables as immutable.
Comments / Resolution	Resolved.

Issue	Typographical Issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below:

	L 9: import {ERC4626} from "openzeppelin-contracts/token/ERC20/extensions/ERC4626.sol"; This import is unused. L 49: uint256 public minterOrderNub; This variable is unused. L 50: uint256 public cpOrderNub; This variable is unused. L 58: mapping(address => mapping(uint256 => uint256)) public userBalancePerOffer; This variable is unused. L 66-90: All events are unused.
Recommendations	Consider fixing the aforementioned typographical issues.
Comments / Resolution	Resolved.

MintEngineExecutor

The MintEngineExecutor contract is the execution element within the MintEngine module. The core logic of this contract is the final execution of minter and rwa offers which then interacts with the DaoCollateral contract for swapping the RWA to STBC. Additionally. The contract handles the decimal normalization and is responsible for the custody of the tokens for the created orders.

Threat Modeling:

Main risks around this contract are based on cross-contract interactions as well as scaling of decimals.

Issue	Incorrect normalization of RWA within executeOrderMinter will result in MinterOffer provider receiving no STBC
Severity	High
Description	The executeOrderMinter function is responsible for matching newly created RWA offers with existing virtual offers (minter offers). The step beforehand matches the RWA offer with any existing minter offers via the poolMatching function in the MinterPool. The returned transferInfo is built as follows: transferInfos = _appendToBytesArray(transferInfos, abi.encode(token, uint96(amountMatched), address(this), uint96(offerId)) At this point, the amountMatched is scaled to 18 decimals and reflects the amount which was matched from both sides, ie. 100 RWA tokens for 100 USDT whatsoever. An inherent assumption is made that these tokens are valued 1:1, an issue therefore has already been created, however, lets now focus on the executeOrderMinter function: (address token, uint96 amount, address owner, uint96 offerId) = abi.decode(transferInfos[i], (address, uint96, address,

	uint96)); amount = uint96(_scaleAmountByDecimalsOut(token, amount)); As one can see, the transferInfo is decoded and the important decoded values are token and amount. What's important now is the scaling for the amount, which happens based on the token (USDT/USDC). If these tokens have 6 decimals (which they do on some networks), the amount is scaled down to 6 decimals. If we follow our example now, this means the amount becomes 100e6. However, the following approval of the RWA to the DaoCollateral contract is made using that exact value and also the swapRwaForStbcMintEngine call is using that value. This will result in the MinterOffer provider receiving 100e6 STBC instead of 1e18.
Recommendations	Consider replacing that approval to uint max within the constructor as well as using the correct amount (1e18 /not normalized) for the RWA.
Comments / Resolution	Partially resolved, the logic has been refactored. However, a new issue has been introduced which is reported in the specific section below.

Issue	Incorrect normalization for minter asset (USDT/USDC) will result in either DoS or inflated distribution of USDT/USDC to the RWA provider
Severity	**High**

Description	The executeOrderCp function is invoked upon executing orders which have been matched in the MintEngineCpOrderbook contract, speaking of a MinterOffer creation which has been matched with existing RWA offers.
	
	Within the orderMatching function in the MintEngineCpOrderbook, the transferInfos array is built as follows:
	
	transferInfos = _appendToBytesArray(transferInfos, abi.encode(offer.token, matched, offer.owner));
	
	which includes the RWA and matched amount.
	
	If we now take a look at the executeOrderCp function, we realize that the normalization happens based on the original RWA decimals:
	
	amount = uint96(_scaleAmountByDecimalsOut(token, amount));
	
	which is perfectly fine for the interaction with the DaoCollateral contract, however, it will result in the USDT/USDC amount being upregulated:
	
	IERC20(stableToken).safeTransfer(owner, amount);
	
	As the RWA provider will now essentially receive 100e18 USDT instead of 100e6

Recommendations	Consider normalizing the minter offer value as well.
Comments / Resolution	Resolved, this has been refactored.

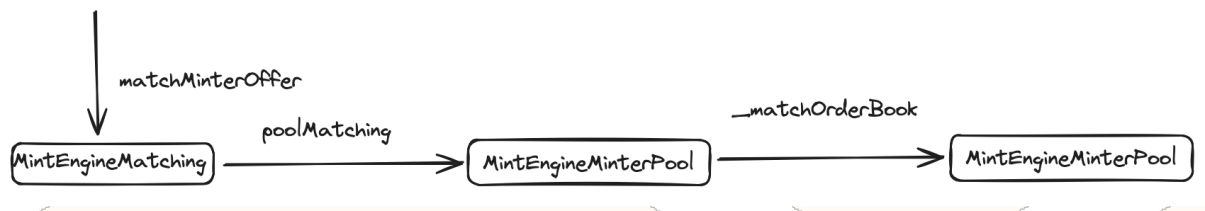
Issue	Unsafe casting from uint256 to uint128 / uint96
Severity	Low
Description	Within the executeOrderMinter and executeOrderCp functions, the elements of bytes within the transferInfo array are fetched, decoded and cached. A problem can arise due to an incorrect casting of the amount to uint128, respectively uint96. The same issue applies for the unsafe casting within the executeOrderCp function: amount = uint96(_scaleAmountByDecimalsOut(token, amount)); Unsafe casting can result in overflows.
Recommendations	Consider simply using uint256 to avoid any unexpected side-effects.
Comments / Resolution	Acknowledged. We still recommend simply keeping it as uint256 throughout all operations, or at least, keep it in the same denomination from the second on where it is fetched from the orderbook.

Issue	Typographical Issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below: L 5: import {ERC20} from "openzeppelin-contracts/token/ERC20/ERC20.sol"; L 13: import {Context} from "openzeppelin-contracts/utils/Context.sol"; This import and the corresponding inheritance can be removed. L 15: import {Queue} from "src/utils/queue.sol"; L 21: import {IMintEngineCpOrderbook} from "src/interfaces/mintEngine/IMintEngineCpOrderbook.sol"; L 36-38: Unused imports.
Recommendations	Consider fixing the aforementioned typographical issues.
Comments / Resolution	Partially resolved.

MintEngineMinterPool

The MintEngineMinterPool contract is mainly responsible for the funds which are corresponding to MinterOffers, such as USDT and USDC and are deposited into the contract upon initiation of minterOffers without corresponding RWA offer or insufficient RWA offers. The contract basically serves as a pooling contract that aggregates any outstanding minterOffers.

The contract incorporates a sophisticated pool matching flow, which is invoked upon the RWA offer creation in the specific scenario that minter offers are existent, it can be illustrated as follows:



The heart of the matching mechanism lies within the _matchOrderBook and essentially implements logic which matches the incoming RWA offer with existent minter offers. This is done on a first-in-first-out basis for the minter offers, based on the ID where the minter offer was created. Minter offers are corresponding to their round specific ID, starting from 1 and increasing every time a RWA offer depletes the minter offers from a specific round. Upon each RWA offer creation and execution, the RWA is swapped for SBTC, which is then added as reward for the minter offers. That way, minter offer creators can simply claim their STBC once the offer was finalized.

It is important to mention that this contract will still contain other bugs with a high likelihood, however, at the current state it is not very time-effective to further search for bugs when the logic needs to be refactored anyways. The allocated time is spent otherwise.

Threat Modeling:

The main risk of this contract is on the correctness of the matching algorithm for several different scenarios. Furthermore, entry functions such as cancelOffer and claim must be specifically observes for their correctness on different input parameters and different time and state specific scenarios.
Due to the high amount of issues, the further audit of the matching mechanism was stopped until the algorithm is fixed.

Issue	Fundamental issue within pool matching mechanism will result in locked minter funds
Severity	**High**
Description	First of all, it needs to be mentioned that in an effort to understand this issue, the whole logical idea of the pool matching mechanism must be understood. On a high-level overview, the pool matching mechanism creates exactly one offer for the balance of a specific ID. What does that mean in detail? Let's assume users have deposited 10_000 USDT for the initial ID, which is ID 1. Upon RWA offer creation, a virtual offer for exactly these 10_000 USDT is created, allowing to exchange RWAs worth 10_000 USDT in index 1. After such a virtual offer has been created, the currentIndex is increased and subsequent users' minter deposits are now allocated to index 2. Once the virtual offer for index 1 has been depleted (ie. RWA worth 10_000) has been exchanged, a new offer is created for index 2, with the amount of pooled funds since the increase to index 2. This mechanism is repeated and repeated. At this point, a lot of respect is due to the usualTech development team, since such a design is quite hard to implement. This comes

naturally with errors. A bug was found in the specific scenario that an RWA order depletes an index but the next index has still sufficient tokens to cover the RWA offer, this condition can be found in Line 338:

if (balanceToken >= remaining)

Now the best way to explain this issue is with a written PoC:

1. Alice created a minter offer with 100 USDT to ID = 1

Storage/memory:

Alice.balance[USDT][1] = 100
USDT.balance[1] = 100
currentID = 1
ERC20Balance = 100

2. Bob creates an RWA offer with a value of 50 USDT
-> matchOrderbook mechanism applies:
- creates initial offer with 100 USDT, since no offers exist
- matches the created offer with provided RWA value and decreases the value by 50
- creates transferInfo with amount of 50 for offerID = 1
- returns with amountMatched = 50 and the corresponding transferInfo
- executes order: adds rewards to ID = 1 and transfers 50 STBC out

Storage/memory:

Alice.balance[USDT][1] = 100
USDT.balance[1] = 100
currentId = 2
ERC20Balance = 50
Offer[1].amount = 50

claimablePerOffer[USDT[[1] = 50
transferInfo[USDT][1] = 50

To this point, everything is great, Alice could already claim partially and there is still an outstanding virtual offer of 50 USDT

3. Charles creates minter offer of 100 USDT for ID = 2

Storage/memory:

Alice.balance[USDT][1] = 100
USDT.balance[1] = 100
currentId = 2
ERC20Balance = 150
Offer[1].amount = 50
claimablePerOffer[USDT[[1] = 50
Charles.balance[USDT[2] = 100
USDT.balance[2] = 100

Note how at this point the balance for ID 2 is 100

4. Derik creates an RWA offer with 60 USDT value
-> matchOrderBook mechanism applies
- match and delete Offer[1], since 60 is larger than 50
- amountMatched = 50
- create transferInfo for offerId 1 with amount = 50
- remaining = 10
- balanceToken = 100
- since balanceToken is larger than remaining, we enter directly in our vulnerable condition:

```
if (balanceToken >= remaining) {
// * We create a new offer with the remaining amount
_createOffer(token, remaining);
transferInfos = _appendToBytesArray(
```

transferInfos, abi.encode(token, remaining, address(this), uint96(++offerId))
);
return (amount, transferInfos);
}

- create new offer for ID = 2 with amount = 10
- create transferInfo for offerId = 2 with amount = 10
- executes order: adds both transferInfos and transfers 60 USDT out

Storage/memory:

Alice.balance[USDT][1] = 100
USDT.balance[1] = 100
currentId = 3
ERC20Balance = 90
Offer[1].amount = 50 <- DELETED
claimablePerOffer[USDT[[1] = 100
Charles.balance[USDT[2] = 100
USDT.balance[2] = 100
USDT.balance[2] = 100
claimablePerOffer[USDT[[2] = 10
Offer[2].amount = 10

Note how the offer for ID = 2 was created with 10 USDT. This is the critical point, since this offer was created, Charles is now prevented from withdrawing his tokens, which is rightfully done so. However, at this point, only 10 claimable tokens are assigned. This will now change in the next step:

5. Erica creates RWA offer with 100 USDT
-> matchOrderBook mechanism applies
- match and delete Offer[2], since it has only 10 USDT allocated
- create transferInfo with ID = 2 and amount = 10

	After this point, no matter what happens, the execution for ID=2 is completed, which means that: *claimablePerOffer[USDT[[2] = 20* which then means that Charles can only withdraw 20 STBC.
Recommendations	One could now argue that the potential solution for this specific condition would be to create the offer not with the remaining leftover of RWA but instead using the balanceToken variable. However, we argue that it is not simply done by that. It is clear that this whole mechanism is of highest engineering complexity and during the development cycle and the first audit round not all specific flows have been unveiled and tested. Specifically, since several other issues are present. It is mandatory to identify all potential call-paths, fix the codebase and execute extensive testing. A simple resolution round will not be efficient to validate the correctness of this complex flow. A re-audit of that flow is necessary.
Comments / Resolution	Resolved, the whole logic has been refactored.

Issue	Fundamental issue within pool matching mechanism will result in highly invalid state which breaks the whole minting engine
Severity	**High**
Description	Similar to the above explained issue, there is another logical flaw existent in the advanced scenarios which will result in a state where the contract has outstanding virtual minter offers without any backed balance, this will then result in a permanent revert if users attempt to create rwa offers. The root-cause of that issue is that an order is created while the transferInfo was filled with the leftover balance, which essentially results in an outstanding virtual order without any backed funds: _createOffer(token, balanceToken); transferInfos = _appendToBytesArray(transferInfos, abi.encode(token, balanceToken, address(this), uint96(++offerId))); PoC: 1. Alice creates minter offer with 200 USDT **storage/memory:** *currentId = 1* *_user[Alice].balancePerOffer[USDT][1] = 200* *_token[USDT].balancePerOffer[1] = 200* *USDTBalance = 200* 2. Bob creates RWA offer with 100 USD value -> virtual offer with 200 USDT is created -> offer is matched and amount is reduced to 100 -> returns with transferInfo and amountMatched

-> executes order

storage/memory:

_user[Alice].balancePerOffer[USDT][1] = 100
_token[USDT].balancePerOffer[1] = 100
USDT.offer.amount = 100 [ID = 1]
currentId = 2
transferInfo[USDT][1] = 100
_token[usUSD].claimablePerOffer[USDT][1] = 100
USDTBalance = 100

3. Charles creates minter offer with 100 USDT for ID = 2
-> executes simple deposit

storage/memory:

_user[Alice].balancePerOffer[USDT][1] = 100
_token[USDT].balancePerOffer[1] = 100
USDT.offer.amount = 100 [ID = 1]
currentId = 2
_token[usUSD].claimablePerOffer[USDT][1] = 100
_user[Charles].balancePerOffer[USDT][2] = 100
_token[USDT].balancePerOffer[2] = 100
USDTBalance = 200

4. Derik creates RWA offer for 250 USD
-> enters matching
- takes offer; deletes offer
- amountMatched = 100
- transferInfo = 100 for ID = 1
- remaining = 150
- balanceToken = 100
- increases amountMatched by balanceToken
- amountMatched = 200

	- creates new offer with 100 USD for ID = 3 - creates transferInfo with amount = 100 for index 2 - order is executed - offer in OrderBook with value of 50 USD is created ***storage/memory:*** *_user[Alice].balancePerOffer[USDT][1] = 100* *_token[USDT].balancePerOffer[1] = 100* *USDT.offer.amount = 0 [ID = 1] <- DELETED* *currentId = 3* *_token[usUSD].claimablePerOffer[USDT][1] = 200* *_user[Charles].balancePerOffer[USDT][2] = 100* *_token[USDT].balancePerOffer[2] = 100* *transferInfo[USDT][1] = 100* *_token[usUSD].claimablePerOffer[USDT][2] = 100* *USDT.offer.amount = 100 [ID = 2]* *transferInfo[USDT][2] = 100* *USDTBalance = 0* We are now in a state where the minterPool has tokens and the OrderBook has an offer. Additionally, the new virtual offer with 100 USD is created for ID = 3, while all tokens have now been transferred out. When now a new RWA offer is created, it will attempt to match this offer with the virtual offer. However, it will always revert since the contract does not have any funds left. This is a highly invalid state and the contract is essentially broken completely
Recommendations	Consider refactoring the matching mechanism and add necessary edge-case tests to prevent such issues from occuring in the future. A full re-audit of the matching mechanism is mandatory and for the MintEngine module in general highly recommended.

Comments / Resolution	Resolved, the logic has been refactored.

Issue	Lack of logical storage update within claim allows for repetitive calls to drain the contract empty
Severity	**High**
Description	The claim function allows users to claim their rightful share of stbc after an rwa to sbtc exchange. The mathematical calculation is as follows: uint256 rewardAmount = _token[usUSD].claimablePerOffer[tokenUsed][id] * stableAmount / _token[tokenUsed].balancePerOffer[id]; Which is simply calculating the proportional amount of the users balance in ratio to the total balance and the claimable amount. The problem is there is no storage update which decreases the user's balance. A repetitive call of the claim function allows for stealing all tokens.
Recommendations	Consider updating the user balance after a claim. It is important to ensure users can only claim once an epoch (currentId) has passed, otherwise users might lose tokens during the claim process. A full re-audit of the whole mintEngine scope is highly recommended.
Comments / Resolution	Partially resolved, the balance is reset but the call can still happen within an active epoch. The ID must not only be validated to be existent but also to have passed.

Issue	Invalid condition check within _cancelOffer will result in multiple downstream issues
Severity	Low

Description	Within the _cancelOffer function, a cancellation fee is applied, which is then distributed to the corresponding bucket for later distribution. The issue with the bucket assignment is an invalid condition check, which first checks if the provided token is USDT and in the scenario where it is not USDT it is automatically assigned towards the USDC bucket:
	
	// * In v0 we only support USDT and USDC
	// TODO: pre-shot the addition of other token
	if (token == USDT) {
	
	_bucketDistribution.addAssetToBucket(BUCKET_DAO_USDT_TREASURY, token, fees);
	} else {
	
	_bucketDistribution.addAssetToBucket(BUCKET_DAO_USDC_TREASURY, token, fees);
	}
	
	As the comment indicates that in V0, only USDT and USDC is supported, therefore this issue is marked as low severity.
	
	However, due to the fact that it is possible to add multiple other tokens:
	
	function createOfferMinter(address token, uint256 amount)
	public
	onlyWhitelistMinterToken(token)
	nonReentrant
	{

	we are the opinion to still mention this issue. If we now assume that any other token besides USDT and USDC is being added, this will result during the cancellation to an assignment of this token towards the USDC bucket which has the following implications: 1. Token is permanently lost 2. Keeper execution will not work anymore since the distribute call with amount = 0 will attempt to distribute the full _balance, however, in fact there is insufficient USDC in the bucket to fulfill the transaction.
Recommendations	Consider switching conditions to explicitly check for USDT and USDC and if none of both conditions are fulfilled, simply send the token to a fee recipient.
Comments / Resolution	Acknowledged.

Issue	getBalancePerToken approach can be manipulated via direct transfers
Severity	Medium
Description	The getBalancePerToken function is used on the following occasions 1. During the order matching mechanism 2. During the createOfferCpToken function to determine if the minter pool is empty. This mechanism can be manipulated via a direct transfer, which can

	then potentially manipulate these operations.
Recommendations	Consider switching to a different accounting mechanism, which cannot be manipulated due to a balance change. It is important that the full mintEngine needs a re-audit.
Comments / Resolution	Partially resolved, this has been refactored but a bug was found.

Issue	Arithmetic operation within claim can be abused to leave users without rewards
Severity	Low
Description	The arithmetic operation within the claim function is as follows: _token[usUSD].claimablePerOffer[tokenUsed][id] * _user[msg.sender].balancePerOffer[tokenUsed][id] / _token[tokenUsed].balancePerOffer[id]; As one can see, this operation has the exact same vulnerability as ERC4626 vaults for the inflation attack. A user can just deposit a large amount of tokens which then rounds rewards for other users down. However, this does not seem like an economically valuable attack, therefore it is just rated as low severity.
Recommendations	Generally speaking, a feasible approach would be setting the balancePerOffer parameter to a virtual value, similar to what OZ does within their implementation. However, this would of course result in less tokens for users due to tokens being allocated to the virtual shares. Given that claimablePerOffer is likely a larger amount, this makes this issue less prone/impossible to happen in practice. Specifically since a claim should only be possible to happen after the provided virtual

	offer is fully consumed.
Comments / Resolution	Acknowledged.

Oracles

AbstractOracle

The Abstract oracle serves as the base oracle implementation which is then inherited by the ClassicalOracle and UsualOracle.

It's main logical mechanism is the getPrice function, which is invoked on the following occasions:

1. DaoCollateral:
-> swap: Determine the USD value of the provided RWA amount

-> swapRWAForStbcMintEngine: Determine the USD value of the provided RWA amount

-> _calculateCoefAndRemainingToMint: Determine the USD value of total RWA backing

-> redeem/redeemDao: Determine the amount of RWA for burned STBC

2. BucketDistribution:
-> mintNewTokens: Fetches the current price of 1 nominal RWA token. Used to compare with the last reported total balance

Compared to traditional oracle implementations, this is the most advanced oracle that BailSec encountered so far. It employs different security mechanisms, namely:

- OracleIsBroken: Checks for faulty return data
- OracleIsFrozen: Checks for oracle staleness

- OraclePriceChangeAboveMax: Checks for unnatural price movements and oracle manipulation

These checks are incorporated within the getPrice function, making it probably the most sophisticated getPrice function which has ever existed. The oracle has 4 different states which will be adjusted dependent on the return value:

- oracleNotInitialized: before initialization
- oracleWorking: upon initialization, as long as return data is correct
- oracleUntrusted: set whenever the oracle has faulty return data, is stale or deviation is too large
- oracleFrozen: set whenever the response is stale

Depending on the current oracle state, the getPrice function flow is changed. At this point it is worth noting that even if the past report was broken, stale or had a large deviation, this will not completely DoS the next calls.

The sophisticated flow of the getPrice function can be distinct based on three different states:

A) oracleNotInitialized: REVERT

B) oracleWorking:

1. Check return data for correctness: set to oracleUntrusted, return lastGoodPrice
2. Check staleness: set to oracleFrozen, return lastGoodPrice
3. Check deviation: set to oracleUntrusted, return lastGoodPrice
4.

If all conditions hold, the most current price is stored and returned.

C) oracleUntrusted:

1. Check if oracle is broken: return lastGoodPrice
2. Check if oracle is frozen: return lastGoodPrice

3. Ensure price deviation between both responses is less/equal 5%: return lastGoodPrice

If all conditions hold, the status will be set to oracleWorking again and the most current price is stored and returned.

D) oracleFrozen

1) Check if the oracle is broken: return lastGoodPrice
2) Check if the oracle is frozen: return lastGoodPrice
3) **Ensure price deviation between both responses is less/equal 5%: REVERT**

If all conditions hold, the status will be set to oracleWorking again and the most current price is stored and returned.

One last additional check upon storing the most current price is a depeg check, which reverts if the asset is a stablecoin and deviates more than 1%.

The price is always stored and returned with 18 decimals.

Threat Modeling:

The main concern here arises around edge-cases which will store an answer while the oracle is frozen/untrusted/broken or external influence which can result in a frozen/broken oracle.

Issue	Incorrect arithmetic operation will result in faulty price difference
Severity	High
Description	The following operation within _bothResponseSimilarPrice aims to calculate the relative price difference as a percentage. uint256 percentPriceDifference = maxPrice - (minPrice) * (DECIMAL_PRECISION) / (minPrice); However, due to the incorrect bracket usage, there's a significant error in the formula, leading to incorrect calculations. The correct calculation for percent difference should divide the difference by the average price or one of the specific prices, and then multiply by DECIMAL_PRECISION. As written, the operation will not yield a percentage difference but rather an incorrect value due to faulty arithmetic logic.
Recommendations	Consider setting the brackets correctly.
Comments / Resolution	Not resolved, the problem still exists, consider adjusting the calculation as follows: uint256 percentPriceDifference = (maxPrice - minPrice) * (DECIMAL_PRECISION) / (minPrice);

Issue	getPrice reverts for faulty return data of _getCurrentOracleResponse
Severity	Medium
Description	The getPrice function was developed in an effort to return the lastGoodPrice whenever the oracle return data is broken. This will however not work because for a broken oracle data the return value for roundId is the following: oracleResponse.roundId = 0; This value will then be used for the _getPrevOracleResponse call, where it attempts to decrease 0 by 1, which will underflow: try priceAggregator.getRoundData(_currentRoundId - 1) returns
Recommendations	There are several solutions for this problem: 1) set roundId = 1 for the scenario of a faulty oracle (this should then also be adjusted in the _badOracleResponse function) 2) Return early within the _getPrevOracleResponse function whenever the provided roundId is zero.
Comments / Resolution	Not resolved, in itself, this will not result in any faulty return data, it will rather revert in the scenario of a broken oracle. However, the protocol was designed in such a way to return the lastGoodPrice for that scenario instead of reverting. We highly recommend fixing this issue.

Issue	Lack of access control for getPrice can be abused to update the oracle state
Severity	Low
Description	Currently, the getPrice function is unguarded while it is meant to be called only by privileged contracts. A direct call can influence the current state and might not be desired.
Recommendations	Consider implementing a corresponding access control mechanism for the getPrice function.
Comments / Resolution	Acknowledged.

Issue	Unconsidered call-path within getPrice can result in revert
Severity	Low
Description	The getPrice function has many different call paths which depend on the past oracle state and the current oracle state. A very specific call-path is the scenario when the past state indicates a frozen oracle. If now the oracle is still frozen or broken, it will return the lastGoodPrice. If it is not frozen and not broken, it is checked if both responses have a similar price, if yes, the new price is stored and returned, if not, the call reverts. In our opinion, the revert scenario was unconsidered for this specific condition and it should rather return the lastGoodPrice.
Recommendations	Consider returning the lastGoodPrice in that scenario that the oracle is not frozen/broken but returns a non-similar price.

Comments / Resolution	Not resolved.

Issue	Return data of lastGoodPrice can be dangerous
Severity	Informational
Description	The getPrice function returns the lastGoodPrice on several occasions where the current oracle return value is broken/stale or incorrect. The lastGoodPrice value as return data can result in issues, specifically if the oracle is untrusted for a longer time, the lastGoodPrice value can differ a lot from the real price.
Recommendations	Consider carefully monitoring the oracle for such situations and eventually pausing the protocol manually if such a scenario occurs.
Comments / Resolution	Acknowledged.

Issue	Truncation when using getQuote function will result in slightly lower return value
Severity	Informational
Description	The getQuote function sets the last 10 decimals from the return value to zero. Currently we are unaware of the idea behind this logic but it seems intended as within the UsualOracle this practice is not executed. However, this fact must still be incorporated in the report.
Recommendations	Consider if there is a rationale behind this, if not consider simply removing the truncation logic.
Comments / Resolution	Resolved, this function has been removed.

Issue	Typographical Issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below: L 81: The modifier is unused.
Recommendations	Consider fixing the aforementioned typographical issue(s)
Comments / Resolution	Acknowledged.

ClassicalOracle

The ClassicalOracle is an extension of the AbstractOracle and is inheriting the said. The only addition to the AbstractOracle is the initializer function that allows for the initial settings as well as the getQuote function for view-only purposes.

Threat Modeling:

The only risk additional to the AbstractOracle is incorrect initialization data

Issue	Assumption that CL oracle returns answer with 8 decimals is incorrect and can lead to faulty lastGoodPrice
Severity	**High**
Description	The initializeTokenOracle function expects the response to be denominated as 8 decimals which is incorrect as it can also be 18 decimals. This can lead to an incorrect lastGoodPrice. Interestingly, this is handled correctly in the rest of the oracle module.
Recommendations	Consider using the decimals from the oracle response.
Comments / Resolution	Resolved.

Issue	initializeTokenOracle function can be called twice, resulting in potential discrepancies
Severity	Low
Description	The initializeTokenOracle function is - as the name already indicates - an initializer function. Typically such functions are only meant to be called once. In the said contract, this function can be called multiple times which violates the rule of initializer functions.
Recommendations	In fact, even if accidentally implemented, such a scenario can be beneficial if the oracle needs to be changed as it can save a new deployment. However, one needs to be careful with this functionality. Consider if this should be fixed or kept as-is.
Comments / Resolution	Acknowledged.

Issue	getQuote function lacks NATSPEC and does not comply with different token decimals
Severity	Low
Description	Presumably, the getQuote function returns the USD amount for a specific token amount, this will not work for tokens with different decimals than 18. This issue has only been rated as low severity for the following reasons: a) The protocol always scales used tokens to 18 decimals b) The getQuote function is not used by the protocol itself

	Therefore, the risk is exposed to third-parties which use this oracle at their own risk.
Recommendations	Consider if it makes sense to implement a normalization for this view-only function.
Comments / Resolution	Partially resolved, the function is now used throughout the architecture to return the corresponding USD price for a specific token and amount. It however only works for tokens with 6 decimals. A universal approach is recommended.

Issue	Lack of validation for timeout variable
Severity	Low
Description	Within the initializeTokenOracle function, the caller can pass a timeout parameter, which determines the time after which the oracle is considered as frozen/stale. This parameter is insufficiently validated, which can result in undesired side-effects if too large/small.
Recommendations	Consider validating this parameter accordingly.
Comments / Resolution	Resolved.

Issue	Typographical Issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below: The following functions are overridden without any purpose: getPrice _oraclePriceChangeAboveMax _oracleLiveAndUnbrokenAndSimilarPrice _bothResponseSimilarPrice _changeStatus _storePrice _storeOraclePrice _getCurrentOracleResponse _getPrevOracleResponse Overriding functions without changing their logic does not serve any purpose.
Recommendations	Consider simply removing these functions, even if marked as virtual in the main contract, these functions do work as expected. *It might happen that the compiler won't allow for this, in that scenario it should be kept as-is.
Comments / Resolution	Resolved.

DataPublisher

The DataPublisher can be considered as a manual oracle which is solely influenced by the usualTech address and whitelisted publishers.

On a high-level overview, the usualTech address can add other addresses as so-called publishers for a specific token. A whitelisted address can therefore update the OracleResponse for the subsequent round with an *arbitrary* data.

Additionally besides removing and adding addresses from/to the whitelist, the usualTech address can block the update for any token, which means that whitelisted addresses can not update the oracleData anymore. The update privilege is then only possible for the usualTech address (assumed it is declared as whitelistedPublisher).

It is important to note that the provided price is always denominated with 18 decimals and the latestRoundData and getRoundData functions are solely invoked by the UsualOracle.

Threat Modeling:

Since the contract state is solely influenced by the governance body, the only issues which can arise are either due to the compromise of keys or due to accidentally incorrect set parameters.

Issue	Governance Risk: Manipulation of OracleResponse
Severity	Governance
Description	Contrary to the ClassicalOracle and UsualOracle where the external data provider is set upon initialization and never changeable, this is not the case for the DataPublisher contract. For the DataPublisher, any whitelisted address can provide the asset price, while the usualTech address can, at any time, add new addresses to the whitelist, opening the possibility for governance privilege abuses.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue	First publishData call should provide two tokenResponsesPerRound
Severity	Low
Description	Currently, the publishData function provides the following oracleData upon calling it: - roundId - answer - timestamp - success

	- decimals Which is perfectly fine. However, as we know, this information is used within the getPrice function in the UsualOracle: OracleResponse memory oracleResponse = _getCurrentOracleResponse(token); OracleResponse memory prevOracleResponse = _getPrevOracleResponse(token, oracleResponse.roundId); Furthermore, we can see and already know, the getPrice function does not only fetch the current response but also the latest response, for security reasons. If now the publishData function was only invoked once, naturally the previous oracleResponse will be faulty, resulting in setting the oracle into the untrusted state. This can be prevented by automatically setting two OracleResponses upon the very first call.
Recommendations	Consider setting two OracleResponse on the init call of publishData.
Comments / Resolution	Acknowledged, this will be done manually.

UsualOracle

The UsualOracle is very similar to the AbstractOracle. It exposes the exact same getPrice logic with all its necessary checks. Additionally it exposes the initializer function, uses a price return data with fixed 18 decimals which removes the necessity for decimal normalization and validates a potential depeg scenario using the TWAP price from a curve pool.

Contrary to the AbstractOracle, the oracle data is not fetched from the AggregatorV3Interface but from the DataPublisher contract, which guarantees that the provided answer is denominated as 18 decimals.

It is important to mention that the used pool address is not provided. Eventual integration issues can occur if the return value for the price_oracle call is in a different denomination or simply wrong.

Threat Modeling:

Similar issues as AbstractOracle can occur, additionally issues due to incorrect data within the DataPublisher or issues related to the depeg mechanism.

Issue	getPrice reverts for faulty return data of _getCurrentOracleResponse
Severity	Medium
Description	The getPrice function was developed in an effort to return the lastGoodPrice whenever the oracle return data is broken. This will however not work because for a broken oracle data the return value for roundId is the following: oracleResponse.roundId = 0; This value will then be used for the _getPrevOracleResponse call, where it attempts to decrease 0 by 1, which will underflow:

	try priceAggregator.getRoundData(_currentRoundId - 1) returns
Recommendations	There are several solutions for this problem: 1) set roundId = 1 for the scenario of a faulty oracle (this should then also be adjusted in the _badOracleResponse function) 2) Return early within the _getPrevOracleResponse function whenever the provided roundId is zero.
Comments / Resolution	Not resolved.

Issue	Lack of validation for timeout variable
Severity	Low
Description	Within the initializeTokenOracle function, the caller can pass a timeout parameter, which determines the time after which the oracle is considered as frozen/stale. This parameter is insufficiently validated, which can result in undesired side-effects if too large/small.
Recommendations	Consider validating this parameter accordingly.
Comments / Resolution	Acknowledged.

Issue	getQuote function lacks NATSPEC and does not comply with different token decimals
Severity	Low
Description	Presumably, the getQuote function returns the USD amount for a specific token amount, this will not work for tokens with different decimals than 18. This issue has only been rated as low severity for the following reasons: a) The protocol always scales used tokens to 18 decimals b) The getQuote function is not used by the protocol itself Therefore, the risk is exposed to third-parties which use this oracle at their own risk.
Recommendations	Consider if it makes sense to implement a normalization for this view-only function.
Comments / Resolution	Resolved, it is now used throughout the architecture.

Issue	initializeTokenOracle function can be called twice, resulting in potential discrepancies
Severity	Low
Description	The initializeTokenOracle function is - as the name already indicates - an initializer function. Typically such functions are only meant to be called once. In the said contract, this function can be called multiple times which violates the rule of initializer functions.

Recommendations	In fact, even if accidentally implemented, such a scenario can be beneficial if the oracle needs to be changed as it can save a new deployment. However, one needs to be careful with this functionality. Consider if this should be fixed or kept as-is.
Comments / Resolution	Resolved, it is now used throughout the architecture.

Issue	Lack of access control for getPrice can be abused to update the oracle state
Severity	Low
Description	Currently, the getPrice function is unguarded while it is meant to be called only by privileged contracts. A direct call can influence the current state and might not be desired.
Recommendations	Consider implementing a corresponding access control mechanism for the getPrice function.
Comments / Resolution	Acknowledged.

Issue	Flaw for checkTokenDepeg logic
Severity	Low
Description	One of the core differences between the UsualOracle and the AbstractOracle is the adjusted depeg check. Within the AbstractOracle the depeg check is depending on the hardcoded price of 1. Whereas for the UsualOracle the depeg mechanism is depending on the curve pool TWAP. However, the checkTokenDepeg function is external and therefore not invoked upon the price store mechanism.
Recommendations	Consider adjusting the function selector to _checkDepegPrice and override the function.
Comments / Resolution	Acknowledged.

Issue	Attacker can manipulate the Curve pool TWAP in an effort to DoS the protocol
Severity	Low
Description	The UsualOracle is using the curve pool TWAP for the depeg check. In the scenario where the oracle returns a value which deviates more than 5% from the Curve TWAP, the getPrice function reverts. The problem lies within the natural manipulation possibility of TWAP oracles, with sufficient economical power an attacker can even manipulate the most secure TWAP oracles. This would then result in a revert of the getPrice function, bricking the protocol. This issue is only present if the aforementioned issue is fixed. Moreover, it is only rated as low since such an attack takes a lot of

	effort and money.
Recommendations	Consider if this could expose a threat to the usualUSD protocol. If yes, consider switching back to the AbstractOracle's depeg check.
Comments / Resolution	Acknowledged.

Issue	Unconsidered call-path within getPrice can result in revert
Severity	Low
Description	The getPrice function has many different call paths which depend on the past oracle state and the current oracle state. A very specific call-path is the scenario when the past state indicates a frozen oracle. If the oracle is still frozen or broken, it will return the lastGoodPrice. If it is not frozen and not broken, it is checked if both responses have a similar price, if yes, the new price is stored and returned, if not, the call reverts. In our opinion, the revert scenario was unconsidered for this specific condition and it should rather return the lastGoodPrice.
Recommendations	Consider returning the lastGoodPrice in that scenario that the oracle is not frozen/broken but returns a non-similar price.
Comments / Resolution	Acknowledged.

Issue	Return data of lastGoodPrice can be dangerous
Severity	Informational
Description	The getPrice function returns the lastGoodPrice on several occasions where the current oracle return value is broken/stale or incorrect. The lastGoodPrice value as return data can result in issues, specifically if the oracle is untrusted for a longer time, the lastGoodPrice value can differ a lot from the real price.
Recommendations	Consider carefully monitoring the oracle for such situations and eventually pausing the protocol manually if such a scenario occurs.
Comments / Resolution	Acknowledged.

Issue	Typographical Issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below: L 5: import {Math} from "openzeppelin-contracts/utils/math/Math.sol"; This import is unused. L 15: USDC

This import is unused.

L 21:

PoolNotFound

This import is unused.

L 36:

using Math for uint256;

This declaration is unused.

L 38:

IDataPublisher public dataPublisher;

This variable can be made immutable.

L 39:

uint32 public timeInterval;

This variable is unused.

L 44:

event AddUsualAssetToPool(address indexed usualAsset, address indexed poolAddress);

This event is unused.

L 47:

	event RemoveUsualAssetToPool(address indexed usualAsset, address indexed poolAddress); This event is unused. L 71: function setTimeInterval(newTimeInterval).. This function is unused. Moreover, all functions which are overridden but not changed in their logic can be simply removed, as overriding without changing the logic does not serve any purpose.
Recommendations	Consider fixing the aforementioned typographical issues.
Comments / Resolution	Resolved.

Registry

RegistryAccess

The RegistryAccess contract is a simple extension of OpenZeppellin's AccessControl library. Contrary to the traditional access control mechanism initialization, no address gets the DEFAULT_ADMIN_ROLE granted, instead, the ADMIN role is considered as the equivalent to the DEFAULT_ADMIN_ROLE.

Upon deployment, the deployer address is not only set as the owner within the Ownable2Step library but also gets the ADMIN role granted.

The different governance privileges can be defined as follows:

1. **usualTech**: Any address with this role can grant and revoke any role to/from any address, including the DEFAULT_ADMIN_ROLE and the ADMIN role. It is important to mention that any address with the DEFAULT_ADMIN_ROLE can still leverage OpenZeppelin's AccessControl library to grant and revoke roles via the grantRole and revokeRole functions, as these functions are not overridden.

2. **Admin**: Any address with this role can invoke the transferAdminOwnership which then sets the pendingOwner to the newOwner parameter. Upon acceptAdminOwnership, the pending owner becomes the new contract owner and ADMIN and the old owner gets the ADMIN role revoked.
The ADMIN can grant and revoke the USUAL_TECH role to and from any address.

Furthermore, the RegistryAccess contract is used throughout the whole architecture, ensuring correct and up-to-date access control.

Threat Modeling:

Registry contracts are generally considered safe from attacks as long as the access control mechanism is working as expected and only trusted parties are added to the governance body.

Issue	addRole and revokeRole functions are not overridden
Severity	Low
Description	As already explained within the description, the DEFAULT_ADMIN_ROLE can still be assigned to any address by the usualTech role. This role can then be leveraged to add and remove roles using the addRole and revokeRole functions from the original AccessControl library.
Recommendations	Consider carefully elaborating if this exposes a risk, if yes consider overriding these functions and revert upon their execution.
Comments / Resolution	Acknowledged.

Issue	Unused view-functions: hasPolMinterRole and hasPolController role
Severity	Informational
Description	Both aforementioned functions are view-only but never used within the architecture. Moreover, they expose the exact same logic but have different function names. This indicates an unfinished development cycle.
Recommendations	Consider removing these aforementioned functions or implement logic for them to be of any use.
Comments / Resolution	Acknowledged.

RegistryContract

The RegistryContract is a very simple registry which allows addresses with the usualTech role to assign addresses to a specific name which consists out of 32 bytes. This name is derived from the hash of the corresponding contract, as a simple example we can take a look in the Constants library:

bytes32 constant CONTRACT_REGISTRY_ACCESS = keccak256("CONTRACT_REGISTRY_ACCESS");

Additionally to the basic registry logic the contract also exposes the UBO mechanism which allows the DAO address to assign one or more addresses to a specific UBO index. This mechanism is for KYC purposes and intends to connect multiple addresses to the same UBO index, which is unique per individual. An UBO with index 0 means an address is not corresponding to a KYC-ed individual, this will result a revert in the corresponding functions such as:

- DaoCollateral.swap
- DaoCollateral._redeemCheck
- DaoCollateral.swapRWAForStbcMintEngine

Which indicates that only KYC-ed individuals can mint and redeem the stablecoin.

Threat Modeling:

Registry contracts are generally considered safe from attacks as long as the access control mechanism is working as expected and only trusted parties are added to the governance body. Furthermore for this contract the getUBO function exposes a potential DoS risk if set incorrect.

Issue	Governance Privilege: Removal of UBO can result in partial loss of funds
Severity	Governance
Description	Within the removeUBO function, the DAO address can remove an UBO identifier from a specific address, resulting in the UBO being set to zero. This will essentially prevent the address from further interaction with the protocol, if for example a RWA offer is existing, this would need to be cancelled by the user, resulting in a loss of the cancellation fee.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue	Lack of validation for name within setContract can result in empty contract name
Severity	Informational
Description	The setContract function allows the usualTech address to assign contracts to their corresponding bytes32 declarations. However, there is no check that the provided name parameter is not an empty bytes32 parameter.
Recommendations	Consider implementing such a check.
Comments / Resolution	Resolved.

Token

ERC20Blacklist

The ERC20Blacklist contract is a simple ERC20 contract which incorporates a blacklist and pausing mechanism. The DAO address can add and remove addresses to/from the blacklist. A blacklisted address can not interact with the contract which means the following:

a) Cannot send or receive tokens

b) Cannot receive tokens via minting

c) Cannot burn tokens

This contract is an abstract contract, meaning it only exists to be inherited. Specifically by:

- Stbc.sol

- usDAO.sol

Threat Modeling:

This contract is a straightforward implementation, risks can occur due to misuse of governance privileges.

Issue	Unnecessary overriding of _burn and _mint
Severity	Informational

Description	Currently, the _burn and _mint functions are overridden in an effort to check for blacklisting. However, this is redundant as already the _beforeTokenTransfer checks for that:
	
	function _beforeTokenTransfer(address from, address to, uint256 amount)
	internal
	virtual
	override
	{
	if (isBlacklisted(from) || isBlacklisted(to)) {
	revert Blacklisted();
	}
	super._beforeTokenTransfer(from, to, amount);
	}
	
	Note that in OZ version 4.9.0 the _beforeTokenTransfer function is invoked before any mint and burn:
	
	mint:
	
	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/09329f8a18f08df65863a5060f6e776bf7fccacf/contracts/token/ERC20/ERC20.sol#L254
	
	burn:
	
	https://github.com/OpenZeppelin/openzeppelin-contracts/blob/09329f8a18f08df65863a5060f6e776bf7fccacf/contracts/token/ERC20/ERC20.sol#L280
	
	Therefore, the _mint and _burn functions can be removed.

	Moreover, the overriding of the _afterTokenTransfer function is unnecessary. It can be possible that the compiler will not accept if these functions are removed, in that scenario, they should be kept.
Recommendations	Consider removing the aforementioned functions, sufficient tests should be executed to ensure correctness. The OZ version should be consistent.
Comments / Resolution	Resolved.

Issue	Variables can be made immutable
Severity	Informational
Description	Variables which are defined within the constructor and never changed afterwards can be marked as immutable, this will include these variables directly into the contracts bytecode, which will save gas whenever these variables are accessed. L 18: IRegistryContract internal _registryContract; L 19: IRegistryAccess internal _registryAccess;
Recommendations	Consider marking these variables as immutable.
Comments / Resolution	Resolved.

ERC20Whitelist

The ERC20Whitelist contract is a ERC20 contract which incorporates a whitelist and pausing mechanism. Contrary to the blacklist contract, only users which are explicitly whitelisted can transfer and receive tokens. This explicitly applies not to minting and burning. Non-whitelisted users can still receive tokens during minting or burn tokens.

This contract is an abstract contract, meaning it only exists to be inherited. In v0, this implementation is not used.

Threat Modeling:

No threats, since not implemented in the overall architecture.

Issue	Variables can be made immutable
Severity	Informational
Description	Variables which are defined within the constructor and never changed afterwards can be marked as immutable, this will include these variables directly into the contracts bytecode, which will save gas whenever these variables are accessed. L 26: IRegistryContract internal _registryContract; L 27: IRegistryAccess internal _registryAccess;
Recommendations	Consider marking these variables as immutable.
Comments / Resolution	Acknowledged.

LsausUSD

The LsausUSD contract is, contrary to all other tokens within the usualUSD ecosystem, almost isolated. This means it is not minted nor burned during any protocol interactions but rather only mintable for external users. The only real interactions that are done with the protocol are the following:

- Transfer-tax is distributed to the LsausUSD bucket

- Distribution of UsDAO from the BUCKET_DAO_LOCKED_SAVINGS_ACCOUNT to itself

- Lock/Burn of usDAO to receive LusDAO within the UsDAO contract

Users can mint this token with STBC and then again burn the token for STBC after a pre-defined bond period, which is determined upon contract deployment. During the time where users are owners of the bond token, they will receive LusDAO as reward, based on a reward algorithm.

Transfers are locked by default but can be unlocked by the DAO address, while upon every transfer, a fee is applied which is distributed to the LsausUSD bucket.

Algorithm explained:

The SAVERS_PERCENTAGE variable is the % of LusDAO reward on the total minted STBC, stretched over the full bonding period. As an example, if 100_000 LsausUSD tokens are minted and the bonding period is 100 days, with a SAVERS_PERCENTAGE of 1%, this means that over the full period, 1000 LusDAO are distributed, respectively each day 10.

Of course this calculation is just a static assumption and due to the dynamic increasing nature of the bond token, the distributed amount will be increasing each day due to the overall increased deposits.

If we now apply the SAVERS_PERCENTAGE of 100 (1%) on the k calculation:

*100 * 1e18 / 1000000 = 1e14*

Now for simplicity purposes, let's assume that 100 days have been passed and the account had every day 100 LusDAO as balance, this is the first and only claim:

yield = k * (nBalance - accumulatedReward[account]) / d;

*1e14 * (100 * 100e18) / 100 = 1e34*

yield /= SCALAR_ONE;

1e34 / 1e18 = 1e16

which perfectly aligns with the intended rate of 1% over the full time period.

Due to the high amount of large-scale severity issues within this contract, it can be assumed that several other issues are still existent as well. These issues can only be tackled one the codebase is refactored and fixed.

It is important to mention that this contract will still contain other bugs with a high likelihood, however, at the current state it is not very time-effective to further search for bugs when the logic needs to be refactored anyways. The allocated time is spent otherwise.

Threat Modeling:

- Circumvent reward logic
- Issues due to transfer
- Issues due to tax
- Draining of STBC

Issue	Multiple bonds at the same time or lack of bond removal can result in DoS attack and loss of funds
Severity	**High**
Description	The yield is distributed/claimed from the BucketDistribution contract: _bucketDistribution.distribute(BUCKET_DAO_LOCKED_SAVINGS_ACCOUNT, yield, abi.encode(p)); This relies on the inherent assumption that sufficient yield is left in the bucket to distribute, which is certainly given due to the incorporated distribution during the STBC minting. A problem will now arise if there is more than one bond active at the same time OR if an old bond has not manually been deactivated within the factory using the removeBond function. A malicious user can simple call the distribute function with the old bond as recipient, this will result in the balance being depleted, since the full balance of the corresponding bucket is always transferred and funds sent to the old contract, which will be stuck there forever. This can specifically happen due to a malicious intent, frontrunning the _claimYield call. This will not only result in a loss of usDAO but also in all STBC being stuck within the LsausUSD contract.
Recommendations	Consider ensuring that never more than one bond is active at the same time and automatically removing expired bonds.

Comments / Resolution	Resolved, this has been refactored.

Issue	Manual distribute call within BucketDistribution will result in locked funds within Bond
Severity	High
Description	The BucketDistribution exposes the distribute function which is publicly callable. A malicious user can simply call this function for the BUCKET_DAO_LOCKED_SAVINGS_ACCOUNT savings account and an existing/running bond. This will then just transfer all usDAO to the bond. The problem is now that there is absolutely no way to retrieve these tokens or burn them to LusDAO for distribution, they are essentially permanently locked. This will not only result in a loss of usDAO but also in all STBC being stuck within the LsausUSD contract.
Recommendations	Consider implementing an access control mechanism for the distribute function. Furthermore ensure that only the LsausUSD contract can call the distribute function for the BUCKET_DAO_LOCKED_SAVINGS_ACCOUNT.
Comments / Resolution	Resolved, this has been refactored.

Issue	Incorrect accounting of accumulatedReward after transfer
Severity	**High**
Description	The transfer function exposes a transfer-tax which results in the recipient getting less tokens than initially intended to transfer. Unfortunately, the accumulatedReward variable for the recipient is increased using the full amount, not the post-tax amount, resulting in a loss of rewards for the recipient: accumulatedReward[recipient] += n * amount; The loss of reward stems from the fact that the accumulatedReward is larger than it should be, resulting in a larger than usual deduction within the _calculateYield function.
Recommendations	Consider using the post-tax amount.
Comments / Resolution	Resolved, this has been refactored.

Issue	Granulation of day can be abused to steal large amount of yield within two blocks
Severity	**High**
Description	The calculation getDaysSinceBondStart will result in a day truncation: return (block.timestamp - bondStart) / 1 days; which means that the day will be shifted after one single block. This methodology can be abused by an attacker to receive a large amount of LusDAO, while only locking STBC for one block.

	PoC: 1. The ending time for the running bond is 1706644800 2. Alice has 1_000_000 STBC and deposits the STBC at TS 1706644799 3. One block is passed and Alice effectively gains yield for 1 day but can unwrap directly in the next block 4. Alice effectively received rewards without locking her tokens This can be done with an arbitrary amount of STBC, receiving an arbitrary amount of LusDAO without any downside of locking tokens.
Recommendations	Consider either changing the granularity to seconds or not allowing deposits one day before the end.
Comments / Resolution	Resolved, this has been refactored.

Issue	Duplicate mint will reset accumulatedRewardDebt
Severity	**High**
Description	The mint function allows users to lock their STBC for a constructor defined period of time, during this time they will obtain each day rewards in LusDAO. Upon the minting the accumulatedReward variable is set to ensure no flash-theft can occur. For subsequent minting, the _calculateYield function is invoked, which will correctly set the accumulatedReward variable for the corresponding account.

	However, at the end of the mint function, the accumulatedReward variable is set again, but only with the newly added amount: accumulatedReward[msg.sender] = n * amountUsUSD; instead of the newly added amount PLUS the old amount. An attacker can abuse this logical flaw to steal tokens from the protocol.
Recommendations	Consider always ensuring that the accumulatedReward variable is set correctly, with the full user-owning balance.
Comments / Resolution	Resolved, this has been refactored

Issue	_calculateYield within mint function is triggered with the wrong parameter, resulting in yield inflation
Severity	**High**
Description	In a similar fashion than the above mentioned issue but not exactly the same, upon the mint function, the _calculateYield function is invoked, with the intent to distribute any remaining fees since the last time. The problem here is that this function is not only invoked with the old balance but also with the newly added balance, which will distribute rewards in hindsight based on the old + new balance. This will also allow anyone to call the mint function at any time during the bond period, immediately gaining yields in hindsight.
Recommendations	Consider calling this function only with the old balance.
Comments / Resolution	Resolved, this has been refactored.

Issue	Lack of upper limit for minting can be abused by malicious user to DoS the full contract
Severity	High
Description	The contract is developed with the natural dynamic of always minting 1% of the distributed STBC amount, while we acknowledge that possibly the correct amount might be minted to this contract upon initial STBC minting (which highly depends on the protocol configuration) one must consider that the same STBC which have been initially minted can be used for multiple bond periods. Therefore, the circulating usDAO amount will naturally not be sufficient to cover the “to be distributed” usDAO/LusDAO, resulting in a full DoS at some point. This flaw will not only result in a DoS during the normal business logic but can also be forced by a malicious attacker.
Recommendations	Consider switching to a more bullet proof mechanism to ensure sufficient coverage of usDAO tokens. An idea might be a privileged minting which is manually executed.
Comments / Resolution	Resolved, this has been refactored.

Issue	Lack of tax application within transferFrom can be used to transfer tokens without paying tax
Severity	Medium
Description	The transfer function exposes a tax, the transferFrom however does not. Users can abuse this inconsistency to transfer the token without a tax.
Recommendations	Consider applying the tax also to the transferFrom function.
Comments / Resolution	Resolved, this has been refactored.

Issue	Lack of emergency withdraw function
Severity	Medium
Description	Currently, there is no emergency unwrap functionality. In the scenario of external reverts, STBC are stuck permanently.
Recommendations	Consider implementing such a function.
Comments / Resolution	Acknowledged, while the logic has been refactored in such a way that an important invariant is that rewards are always covered, which would prevent potential rewards in the reward update and transfer mechanism, we are still of the opinion that an emergencyWithdraw function should be implemented (for users), which simply ignores about the reward update (but still resets the lastRewardBlock).

LusDAO

The LusDAO token is the illiquid, locked version of the UsDAO. It serves as the main governance token, inheriting OpenZeppelin's ERC20Votes extension.

Within the global architecture, minting happens on the following occasions:

1. During burning UsDAO within the lock function in the UsDAO contract
2. During burning UsDAO within the lock function in the UsualTokenWrapper, this logic will be removed as it is already existent within the UsDAO token.
3. During the reward distribution to individuals (rwaProvider, stableProvider) after the MintEngine execution

A logical burn is solely happening within the burnAsset function in the BucketDistribution contract, which means that at this point, only governance can unlock LusDAO to usDAO. Additionally, users can directly burn their tokens without any benefit.

The LusDAO token solely serves as a governance token, which is implemented in V1 and is only obtainable via burning usDAO, as a minting reward for using the MintEngine mechanism or via the bond mechanism.

Additionally, one may obtain it as reward from the curve implementation. It is non-transferrable for the majority of addresses, since the transfer is only intended for addresses with the LUSDAO_TRANSFER role.

*The ERC20Votes extension is out of scope and the assumption is made that this works as expected.

Threat Modeling:

Issues can occur due to incorrect usage of ERC20Votes. However, since that is not used within V0, it is a topic for V1.

Issue	Certain variables can be made immutable
Severity	Informational
Description	Variables which are defined within the constructor and never changed afterwards can be marked as immutable, this will include these variables directly into the contracts bytecode, which will save gas whenever these variables are accessed. L 28: IRegistryAccess internal _registryAccess; L29: IRegistryContract internal _registryContract;
Recommendations	Consider marking these variables as immutable.
Comments / Resolution	Acknowledged.

Issue	Typographical Issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below:

	L 25: using SafeERC20 for ERC20; This declaration is unused. L 26: using SafeERC20 for ILusDAO; This declaration is unused. L 56: modifier onlyUsualTech() This modifier is unused.
Recommendations	Consider fixing the aforementioned issues.
Comments / Resolution	Acknowledged.

Stbc

The Stbc contract is the stablecoin which is used by the protocol and can also be identified as usUSD. While this contract is deployed via the StbcFactory, it is notable that the protocol currently only incorporates one Stbc, not multiple. Leveraging the external RegistryAccess contract, two privileged roles are exposed:

STBC_MINT: Allows the role owner to mint STBC

STBC_BURN: Allows the role owner to burn STBC

Within the global architecture, minting happens on the following occasions:

1. **BucketDistribution:** Upon the mintNewTokens function, whenever the RWA collateral becomes more valuable the delta between the old value and new value is minted towards the contract, increasing rwaInfo.stbcBacked.

2. **DaoCollateral**: Upon the swap function (casual minting of STBC with RWA).

3. **DaoCollateral**: Upon the MintEngine execution during the deal execution.

4. **DaoCollateral**: Upon a redeem, minting the fee to the bucket.

Whereas burning happens on the following occasions:

1. **DaoCollateral**: During the redeem of STBC, the full amount of STBC is burned.

For decentralization purposes it is important to ensure that only the necessary contracts have the corresponding roles granted.

Threat Modeling:

Issues can occur due to the incorrect usage of governance privileges, such as blacklisting innocent users.

Issue	Typographical Issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below: L 5: import {SafeERC20} from "openzeppelin-contracts/token/ERC20/utils/SafeERC20.sol"; This import is unnecessary. L 6: import {ERC20} from "openzeppelin-contracts/token/ERC20/ERC20.sol"; The explicit import of the ERC20 contract is redundant, since the ERC20Blacklist import already naturally includes ERC20. L 17: using SafeERC20 for ERC20; This declaration is unused.
Recommendations	Consider fixing the aforementioned issues.
Comments / Resolution	Acknowledged.

usDAO

The usDAO token is, similar to the Stbc token, an ERC20 token which inherits the blacklist mechanism.

There are three scenarios how usDAO is minted:

1. As protocol reward during the mintNewToken logic within the BucketDistribution contract, based on the delta value and the usDAORatio.

2. As protocol reward during minting of STBC via swap and swapRWAForStbcMintEngine.

3. Via the daoTokenUnwrap function within the DaoCollateral to unlock LusDAO tokens in the bucket for usDAO.

There is only one scenario where usDAO is burned: upon the lock call which burns usDAO and mints the same amount of LusDAO.

Therefore, the authorized minters should only be the BucketDistribution and DaoCollateral contract. Authorized burner respectively the UsualTokenWrapper.

The usDAO token is the governance token of the usualUSD protocol in its liquid and transferable state.

Threat Modeling:

Issues can occur due to the incorrect usage of governance privileges, such as blacklisting innocent users and an absence of minter privileges for the LusDAO token.

Issue	Lack of access control during burnFrom allows for burning tokens from arbitrary addresses
Severity	High
Description	The burnFrom function usually allows a privileged caller to burn tokens from a specific account. However, the access control is non-existing here, allowing anyone to burn tokens from anyone. This will not only result in users losing tokens but also the UniswapV2 pairs to be drained completely due to a slightly advanced exploit, similar to the SafeMoon hack last year which occurred due to an un-audited proxy upgrade, which introduced this vulnerability.
Recommendations	Consider implementing a proper access control mechanism for this function.
Comments / Resolution	Resolved.

Issue	Typographical Issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below: L 5: import {SafeERC20} from "openzeppelin-contracts/token/ERC20/utils/SafeERC20.sol"; SafeERC20 is not used within this token contract. L 6: import {ERC20} from "openzeppelin-contracts/token/ERC20/ERC20.sol"; ERC20 is already inherently imported by the ERC20Blacklist import. L 19: using SafeERC20 for ERC20; SafeERC20 is not used within this token contract.
Recommendations	Consider fixing the aforementioned issues.
Comments / Resolution	Acknowledged.

Utils

Queue

The Queue contract contains the logic for handling RWA offers, which is not limited to RWA offers in the OrderBook but also meant to be used for virtualOffers within the MinterPool contract. Therefore, even though imported in several contracts, this library is solely used within the MintEngineCpOrderbook and MintEngineMinterpool contracts.

The core functionalities are creating, taking and canceling offers as well as view-only functions.

Threat Modeling:

Potential issues for this contract are within the integration towards MintEngineMinterPool and MintEngineCpOrderbook

Issue	Unbounded loops can result in revert due to excess gas consume
Severity	Informational
Description	Within the getOfferFromId, get and contains functions, a loop over the whole queue is executed. Such unbounded loops are risky as they can run out of gas if the array grows too large. However, due to the fact that there are no expensive gas operations executed within the loop and because the getOfferFromId function is only invoked for RWA orders which have a minimum value of 10_000 USD, this issue seems rather theoretical than practical.
Recommendations	Consider if this issue will expose a risk for your protocol, if yes the whole mechanism needs to be refactored into an enumerableSet-like mechanism.
Comments / Resolution	Acknowledged.

Issue	Unused functions: back, getOwnerFromId and clear
Severity	Informational
Description	These aforementioned internal functions are unused and will therefore never get executed. The implementation of functions which are never used unnecessarily increases the contract size and confuses third parties.
Recommendations	Consider removing the aforementioned functions.
Comments / Resolution	Acknowledged.

Issue	TAKEN condition within take function will be never met
Severity	Informational
Description	The take function is responsible for emptying/deleting or adjusting offers. If the requested amount is larger or equal than the offer.amount, then the offer will be deleted from the queue. If the offer.amount is larger than the requested amount, then the offer.amount is just decreased accordingly but not deleted. At the beginning of the function there is a check if the offer is TAKEN or CANCELED, which will the automatically delete the offer. However, the condition of the order being TAKEN but still being fetched during the take call is simply impossible, since it is directly deleted whenever marked as TAKEN.
Recommendations	Consider removing that redundant check.

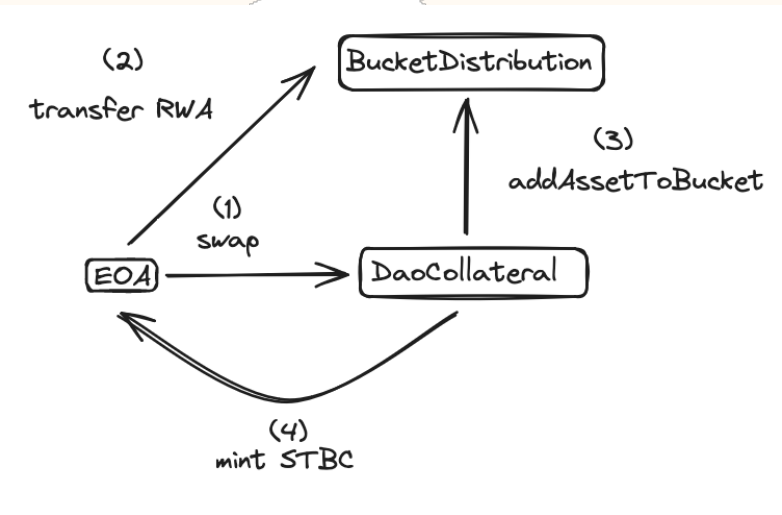
Comments / Resolution	Acknowledged.

Core

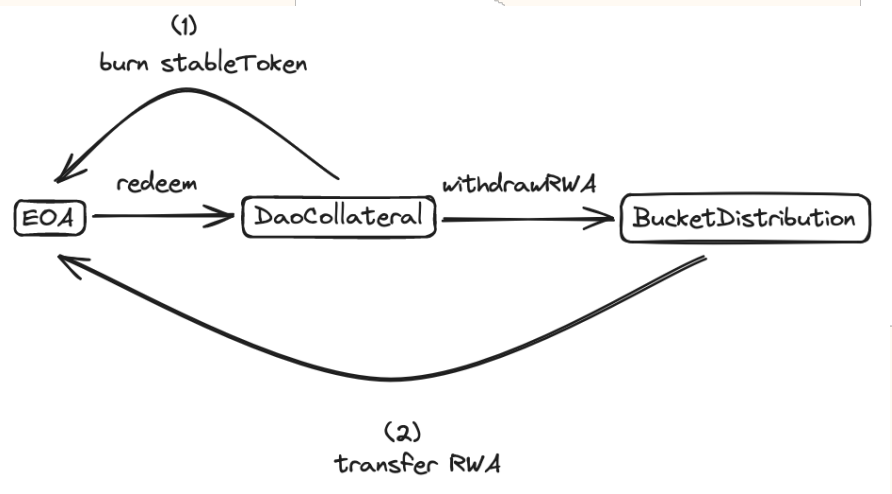
DaoCollateral

The DaoCollateral contract exposes the core of the whole protocol, users can directly interact with this contract to swap (mint) STBC tokens with the corresponding RWA amount and redeem STBC to receive the corresponding RWA amount. The redeeming of STBC incurs a redeemFee which can be freely settable by the DAO address.

The swap flow without fee and rewards can be illustrated as follows:



The redeem flow without fee can be illustrated as follows:



In addition towards the direct interaction by users, the contract also exposes functionality for the MintEngine to interact with it, which is handled within the swapRWAForStbcMintEngine function.

The contract exposes a sensitive usDAO minting structure, which automatically mints usDAO upon any swap or “swap and mint” interaction.

For the swap interaction, usDAO is minted to the bucket and allocated towards different buckets, namely:

1. BUCKET_MM_USDAO_LOAN

2. BUCKET_DAO_LOCKED_SAVINGS_ACCOUNT

3. BUCKET_DAO_LP_SAVINGS_ACCOUNT

4. BUCKET_DAO_AFFILIATION
5. BUCKET_DAO_BONUS

6. BUCKET_DAO_LOTTERY

7. BUCKET_DAO_USDAO_TREASURY

The distribution ratio for these buckets can be set by the DAO address via the setDistributionRatio function.

For the swapRwaForStbcMintEngine interaction, the amount is not only distributed to the above-mentioned buckets but also a share is distributed towards the stable and rwaProvider, as lusDAO. It is additionally important to mention that the architecture, as-is, exposes a specific vulnerability due to the fact that users will receive rewards upon “swap and mint” and potentially pay a lower fee upon redeem that received as minting reward, denominated in USD.

To counter a potential exploit where users repetitively execute the “swap and mint” logic and then redeem the STBC again, a redeemBalance variable was exposed, which is increased upon each redeem. This variable is then counter-checked within the swapRWAForStbcMintEngine function to ensure it is zero. Further it must be noted that the redeemBalance is not assigned to a single address but rather a so-called UBO identifier, which ideally is linked to a KYC-ed individual.

Due to the dynamic price action of RWAs, the possibility exists that the amount in the RWA bucket does not cover the stablecoin circulation anymore, which would naturally result in a depeg of the stablecoin. To counter this issue, the so-called “CBR mode” has been implemented and can be activated and deactivated by the DAO

address. The CBR mode has the following effects on the protocol:

a) Prevents minting new STBC for the protocol profit.

b) Determines the amount of RWA which is received upon STBC redeeming.

For case b), a specific coefficient is calculated upon CBR mode activation, which is based on the following parameters:

1. Total USD value of all underlying RWAs

2. Bucket Insurance Fund

3. STBC supply

This coefficient is then used to calculate the received RWA amount for a specific amount of STBC.

Threat Modeling:

This contract involves multiple different user entries: Minting, redeeming or swapping via MintEngine, inherent risk comes from all different state transitions which can be triggered by users. Multiple different arithmetic calculations with regard to correct redeem/burn/mint amounts and reward calculations must be carefully validated. The use of the CBR mode comes with inherent risk, which is not only related to the correctness of the implementation but also to MEV. This contract is only compatible with one STBC and its corresponding RWAs. It will not work for multiple different STBCs.

Issue	Combination of dynamic price nature from lusDAO token and bypass of _redeemBalance increase can be used to profit from repetitive MintEngine execution and STBC redeems
Severity	**High**

Description	The _getUsDAORewards function is invoked upon the swap and swapRWAForStbtcMintEngine function and calculates the amount of usDAO rewards to be distributed towards the different buckets and eventually the amount of LusDAO to the rwaProvider and stableProvider, which is only the case for the swapRWAForStbcMintEngine flow.
	
	The swapRWAForStbcMintEngine function is using the _redeemBalance of the specific RWA provider. This is specifically important for users once this function is invoked via the swapRWAForStbtcMintEngine function as this means the rwaProvider and stableOwner will both get a part of the reward, minted as LusDao.
	
	For the swap function, this logic is negligible, since the rwaProvider and stableProvider are address(0) anyways, hence only governance will receive the reward, no abuse can happen here.
	
	Now, it is important to understand that this variable has a negative impact on the amount of usDAO rewards that are minted:
	if (redeemBalance > 0) {
	if (amount > redeemBalance) {
	amountToReward = amount - redeemBalance;
	_redeemBalance[ubo] = 0;
	} else {
	_redeemBalance[ubo] = redeemBalance - amount;
	return 0;
	}
	}

and specifically **it does not allow for invoking the swapRWAForStbcMintEngine function** if the redeemBalance is > 0:

```
// can't swap if user has a redeem balance
if (_redeemBalance[ubo] > 0) {
revert NotEnoughDeposit();
}
```

As already mentioned in the description, this mechanism was implemented to prevent any repetitive swap/mint and redeem calls which might allow for an unfair advantage.

Now as next step, it is important to understand on what occasion the _redeemBalance is exactly increased:

```
_redeemBalance[ubo] += amount;
```

which is trivially during the redeem of a STBC.

So now we can observe that a simple transfer to another KYC-ed address allows for the redeem with the other address' UBO code, which means that the _redeemBalance can be increased using a "unused" address, which can be certainly any kyc-ed user.

Now we can think of an exploit which allows a user to mint/swap and redeem without increasing the _redeemBalance, resulting in a potential economical gain.

PoC:
1. Alice, Bob and Charles are friends, all of them are KYC-ed individuals but they crafted a "profitable trading strategy" *cough*.

2. Alice creates a RWA offer with 100_000 USD value, which is pushed in the MintEngineCpOrderbook.

3. Bob creates a minter offer with 100_000 USDT.

4. These offers are now matched in the MintEngineCpOrderbook, upon the minter offer creation.

5. The swapRWAForStbcMintEngine function is invoked with Alice as rwaProvider and Bob as stableProvider. Alice and Bob get a certain amount of lusDAO, since the lusDAO price is currently high, this amount is larger than 50 BPS of the 100_000 USD, remember, the lusDAO price is dynamic and the minting amount is not adjusted during every single block.

6. Alice receives the USDT and Bob receives STBC

7. Bob transfers STBC to Charles, Charles redeems the STBC and transfers the RWA back to Bob.

8. Bob now has the RWA and Alice has the USDT, both have a _redeemBalance of zero.

9. This pattern is repeated until the price of lusDAO decreases such that this attack is no longer profitable. The protocol was effectively tricked.

The root-cause for this issue is the fact that the _redeemBalance increase can be tricked via a simple transfer combined with the dynamic price nature of the lusDAO token.

At this point, it is worth pointing out the calculation behind the lusDAO minting amount:

```
        // User have lusDAORewardRatio bps (e.g 0.5%) of the
amount of usUSD in usdao
        // equivalent to amount * lusDAORewardRatio /
```

SCALAR_TEN_KWEI * 1e18 / usdaoPrice
        userReward = amount * lusDAORewardRatio *
SCALAR_HUNDRED_SZABO / usdaoPrice;

First of all, the comment indicates that the lusDAORewardRatio could be 0.5%, which is exactly the redeemFee. Secondly, we can see that the calculation is adjusted based on the usDAOPrice, which means that the fee is set to such a value which puts the usDAOPrice in reflection to meet exactly 0.5%. Whenever the usDAOPrice is increased, this means the reward will be decreased, keeping an invariant of 0.5% fee. However, since usDAO != lusDAO and there is no guarantee that both prices correlate 1:1, the possibility exists that lusDAO increases more than usDAO, while the usDAOPrice is still used as a divisor, resulting in a higher fee like 0.5%.
While this is theoretically possible, one also need to mention that (whichever mechanism might be used for lusDAO price finding) a price increase of lusDAO combined to usDAO will be countered by users minting lusDAO with usDAO.

Furthermore, the lusDAO token is a locked token, which means that it cannot be sold on secondary markets, hence this attack can just result in a “paper-gain” at this point.

Due to these facts and the current V0 status, the impact of this issue is rather low, but depending on V1 developments, it might be desired to revisit this issue and potentially lower the reward fee.

*This issue was bumped back to high after a revisit of the _mintLusDAO function:

        // User have lusDAORewardRatio bps (e.g 0.5%) of the amount of usUSD in usdao
        // equivalent to amount * lusDAORewardRatio /
SCALAR_TEN_KWEI * 1e18 / usdaoPrice
        userReward = amount * lusDAORewardRatio *
SCALAR_HUNDRED_SZABO / usdaoPrice;

	setting of lusDAORewardRatio variable: cpLusDAOReward = _cpLusDAOReward; // BPS for the collateral reward 50BPS = 0.5% which indicates that in our example, both Alice and Bob will get 0.5% each on the usdAmount, while the redeem of the equivalent USD value will only cost 0.5%, hence a 0.5% gain is achieved, even though the token is locked, this still seems like an issue. This attack can be either executed via repetitive transactions, a flashloan or with large private capital. If used with a flashloan, the _redeemBalance issue does not even necessarily be present, however, depending on the flashloan fee, this attack might be less economically profitable.
Recommendations	Consider simply ensuring that the reward for the rwaProvider and stableProvider together does not exceed 0.5%.
Comments / Resolution	Acknowledged, the team indicated they will handle the fee in such a way that it is not economically exploitable. Additionally, they communicated that they are re-thinking the design of the reward distribution. The further inspection will be subject to Spearbit.

Issue	Incorrect arithmetic operation can result in revert of swapRwaForStbcMintEngine
Severity	**High**

Description	Contrary to the swap function, the swapRwaForStbcMintEngine function not only mints a fee to the protocol but also to the rwaProvider and stableProvider. This is triggered by the provides parameters to the _distributeUsDAO function:
	
	_distributeUsDAO(amountUsDAOToMint, priceInUSD, rwaProvider, stableOwner);
	
	if (stableOwner != address(0) && rwaProvider != address(0)) {
	IOracle oracle = IOracle(_registryContract.getContract(CONTRACT_ORACLE_USUAL));
	uint256 usdaoPrice = oracle.getPrice(address(_usdao));
	
	userReward = _mintLusDAO(usdAmount, cpLusDAOReward, usdaoPrice, rwaProvider);
	userReward += _mintLusDAO(usdAmount, minterLusDAOReward, usdaoPrice, stableOwner);
	}
	
	As one can see, a calculation is being executed for the minted amount of LusDAO, this calculation is based on the swapped usdAmount.
	
	The issue is now that this userReward is later deducted from the amount variable:
	
	uint256 bucketAmount = amount - userReward;
	
	The problem lies here in the calculation of the amount variable, which is already executed within the swapRwaForStbcMintEngine function, specifically here:

	uint256 usDaoRewards = _getUsDAORewards(amountInUSD, ubo); we quickly notice that the _redeemBalance influences this variable and can potentially result in a very small amount variable, depending on the circumstances. However, the userReward is *not* depending on the _redeemBalance and can therefore be larger than the amount variable. This will result in an underflow and therefore a revert of the whole MintEngine flow.
Recommendations	This issue is a textbook example of comparison/arithmetic operations for/with two unrelated values. Consider following a clear structure to ensure both reward calculations are based on the same logic. Optionally, a simple check can be executed which then simply returns early in the scenario where the userReward > amount. However, this would not fix the root-cause of the issue but just the symptom. **On a different note**: Since several issues are related to the _redeemBalance logic and the contract can simply be configured in such a way that individuals will never receive more value than they will lose again during the redeem (0.5%), it might make sense to think about completely removing the _redeemBalance logic.
Comments / Resolution	Resolved, the userReward is now a pro-rata product with the same base as the amount. The only problem could occur whenever the shares are higher, which would then result in an underflow. A corresponding issue has been created and recommendations were given.

Issue	BUCKET_INSURANCE_USD is non-existent
Severity	Medium
Description	The BUCKET_INSURANCE_USD bucket is used for the CBR mode with the intention to fully or partially fill holes in the STBC backing. However, there is no such flow that in fact distributes USD towards this specific bucket.

Recommendations	Consider specifically adding towards this bucket during the specific actions. We highly assume that the minter offer cancellation and the corresponding USDT and USDC fee should flow towards this bucket:
	
	if (token == USDT) {
	
	_bucketDistribution.addAssetToBucket(BUCKET_DAO_USDT_TREASURY, token, fees);
	} else {
	
	_bucketDistribution.addAssetToBucket(BUCKET_DAO_USDC_TREASURY, token, fees);
	}
	
	However, as one can see, they are assigned to different buckets. This should be changed.
	
	Theoretically it is also possible to set assign the BUCKET_PROVIDER role to an EOA and then manually transfer tokens to the BucketDistribution contract and add them towards the BUCKET_INSURANCE_USD bucket, however, we are still of the opinion that this should happen automatically to further increase trustlessness.

Comments / Resolution	Acknowledge, this bucket will be created and filled manually.

Issue	Lack of validation within activateCBR can result in draining of protocol
Severity	Medium

Description	The activateCBR function allows the DAO address to activate the CBR mode which includes a coefficient calculation. The problem with this flaw is that there is no check which ensures that the coefficient is <= 1e18.
	
	In such a scenario where the underlying funds plus the insurance fund divided by the totalSupply of STBC is > 1e18, this will result in a vulnerability where users can mint STBC and then redeem it for more than they have provided during the minting:
	
	if (isCBROn) {
	amountInToken = amountInToken * cbrCoef / SCALAR_ONE;
	}

Recommendations	Consider explicitly validating that the calculated cbrCoef is <= 1e18.
Comments / Resolution	Resolved, such a check has been implemented within the _calculateCoefAndRemainingToMint function.

Issue	CBR mode only representing a snapshot of the conversion rate at the point of activation can result in users receiving more/less tokens than desired.
Severity	Medium
Description	During the activation of the CBR mode, the current underlying USD value is being determined and then used for the coefficient calculation. However, this is just a static assumption, in fact the underlying values can directly increase/decrease afterwards. This can result in users accidentally receiving less or more tokens than the most actual conversion rate.
Recommendations	A potential solution would be to calculate the coefficient once per block during the redeem call, this would then properly reflect the current conversion rate. However, one need to be cautious during the implementation to ensure users cannot manipulate the conversion rate (ie. manipulating the insurance fund, whatsoever).
Comments / Resolution	Acknowledged.

Issue	Lack of impact from _redeemBalance on LusDAO amount will mitigate intended exploit prevention
Severity	Medium

Description	As already mentioned and also confirmed by the team, the _redeemBalance logic was implemented in an effort to counter repetitive calls to gain an economical advantage due to the reward minting upon MintEngine execution. Simply spoken, the _redeemBalance will decrease the distributed rewards.
	
	This is true, but only partially: The usDAO rewards are determined as follows:
	
	uint256 amountUsDAOToMint = _getUsDAORewards(amountInUSD, ubo);
	
	which is then using the _redeemBalance to potentially decrease the usDaoRewards parameter. The issue here is that this is not the variable which is used for the individual stableProvider and rwaProvider reward distribution, if we take a look at the reward calculation for these individuals:
	
	// if stableOwner and rwaProvider is not address(0) then it is an indirect swap
	if (stableOwner != address(0) && rwaProvider != address(0)) {
	IOracle oracle = IOracle(_registryContract.getContract(CONTRACT_ORACLE_USUAL));
	uint256 usdaoPrice = oracle.getPrice(address(_usdao));
	
	userReward = _mintLusDAO(usdAmount, cpLusDAOReward, usdaoPrice, rwaProvider);
	userReward += _mintLusDAO(usdAmount, minterLusDAOReward, usdaoPrice, stableOwner);
	}

	we quickly realize that in fact only the usdAmount is important for the calculation and the amountUsDAOToMint is just used for the protocol rewards, which underlies full governance privileges anyways.
Recommendations	Consider re-evaluating this mechanism and applying changes based on the intended functionality. In our opinion an economical exploit is prevented as long as redeemFee > userReward
Comments / Resolution	Acknowledged, the team indicated that a new reward mechanism is in plan.

Issue	Sudden activation of CBRMode can result in loss of funds
Severity	Medium
Description	The CBRMode can be activated by the DAO address via the activateCBR function. The possibility exists that this call can be executed directly before a user attempts to redeem tokens. This will then result in an unexpected loss for the user.
Recommendations	Consider implementing a minimumOut parameter for the redeem function, which at the end of the function can be checked against the returnedCollateral variable.
Comments / Resolution	Acknowledged.

Issue	Sub-optimal CBR logic can result in total loss of funds for users
Severity	Medium
Description	The CBR mode must be manually activated via activateCBR and only afterwards the CBRCoef will handle how much RWA will be received from a redemption. The problem now lies within the manual nature of this mechanism, once a decline of the RWA value is noticed, a user can simply redeem his STBC, specifically if that user owns a majority of the STBC, he can withdraw his share for a 1:1 ratio (ignoring the redemption fee). If now the STBC is already insufficiently backed, the other users will bear the loss, which can be up to a total loss, depending on how large the backing hole is. It goes without saying that users can also simply frontrun the activateCBR call. This issue is further accelerated due to a constant STBC protocol minting whenever the RWA increases in value. If these minted tokens are not used in a smart manner, even the smallest decline will result in a backing hole, since there is no profit buffer. Consider the following PoC: 1. Alice deposits 100 RWA = 100 USD -> receives 100 STBC 2. Bob deposits 10 RWA = 10 USD -> receives 10 STBC 3. RWA value drops 9% to 0.91 USD/RWA 4. Alice withdraws 100 STBC -> receives 109,8 STBC 5. Protocol is insolvent, Bob will not receive any tokens back. This issue is inherently present if the RWA price decreases for each unique user's deposit, this can also be abused for users to create delta-neutral positions, as the code always allows for receiving the

	exact same USD value back (deducted the 0.5% fee and assumed the CBR mode is no activated)
Recommendations	Consider re-thinking this mechanism, while we agree that it might not be desired to implement an automatic check for the redemption due other potential occurring exploit mechanisms from such an implementation, we highly recommend strictly keeping these profits secure to ensure any potential RWA value increase brings a protocol buffer. If these additional minted STBC are redeemed, this will accelerate that issue. Moreover, specifically the activateCBR call should be done using a protected rpc to prevent from getting frontran and the totalSupply of the STBC should be passed as input parameter and cross-checked once the function is triggered.
Comments / Resolution	Acknowledged, no change to this mechanism has been made. We are however still of the opinion that this can expose a significant systematic risk to the system. Therefore, it might make sense to implement an additional mechanism which prevents users from redeeming whenever the RWA value has declined by a specific threshold / the backing in $ terms has decreased from a 1:1 backing goal, this could be 2.5% as example. This could basically mean an additional loop before each redeem to fetch the totalRwaValueInUSD and revert if this is not at least 97.5% of the overall STBC circulation. With such a mechanism, the protocol could essentially prevent itself from being arbitraged (users redeem STBC for the full amount, while in reality the backing is not full)

Issue	Lack of state update for RWA bucket during setUsDAORatio will result in reward manipulation in hindsight
Severity	Low
Description	The setUsDAORatio function allows the DAO address to update the usDAORatio, which is used upon two occasions: 1. To determine the amount of UsDAO rewards to the protocol, respectively LusDAO to the stable and RWAProvider. 2. To determine the emission rate for usDAO during the _mintNewToken function: uint256 usdaoEmissionRate = IDaoCollateral(registryContract.getContract(CONTRACT_DAO_COLLATERAL)).usDAORatio(); The update will change the usdaoEmissionRate in hindsight, leading to a falsified emission rate.
Recommendations	Consider updating the RWA bucket (calling mintNewToken on the BucketDistribution contract) before the update of the usDAORatio.
Comments / Resolution	Acknowledged.

Issue	daoTokenUnwrap function can never be executed
Severity	Low
Description	The daoTokenUnwrap function unwraps the LusDAO (locked) to usDAO (unlocked), which is a significant business logic step. However, currently it does not seem as if there is a way for LusDAO to be distributed to a bucket. The team indicated that in a newer commit this issue is already fixed.
Recommendations	Consider implementing an automatic mechanism to incorporate a LusDAO distribution to the lusDAOBucket.
Comments / Resolution	Acknowledged.

Issue	Lack of validation for all variables
Severity	Low
Description	The following variables lack validation: redeemFee, usDaoRatio, minterLusDAOReward, cpLusDAOReward and distributionRatio. If set incorrect, this can result in loss of funds and DoS.
Recommendations	Consider validating these not only in their corresponding setter function but also within the constructor.
Comments / Resolution	Resolved

Issue	redeemDao function credits _redeemBalance to DAO address
Severity	Low
Description	Within the redeemDao function, the DAO address can redeem a STBC for the corresponding RWA, however, similar to the standard redeem call, the _redeemBalance for the DAO address is increased. This does not seem like an appropriate mechanism for a privileged function and might result in downstream issues for the DAO address if there is an existing _redeemBalance.
Recommendations	Consider simply ignoring the _redeemBalance accounting for the DAO address.
Comments / Resolution	Acknowledged.

Issue	_calculateCoefAndRemainingToMint can run out of gas if the RWA array grows too large for one STBC
Severity	Low
Description	The _calculateCoefAndRemainingToMint function iterates through associated RWAs, computes their values in USD, and aggregates them. However, if the array of corresponding RWAs grows too large, the loop will run out of gas. This issue was only rated as low severity since in practice there will likely not be 100+ RWAs added to one STBC.
Recommendations	Consider adding a reasonable upper limit within the TokenMapping contract as how large the _stbcToRwaLastId can grow.

Comments / Resolution	Resolved, the array size is limited to 10.

Issue	Payable keyword for selfPermit function can result in loss of ETH
Severity	Low
Description	The selfPermit function incorporates the payable keyword which allows users to attach a msg.value. However, if a user accidentally attaches any ETH with the call, this ETH will be permanently stuck.
Recommendations	Consider removing this keyword.
Comments / Resolution	Resolved.

Issue	_getQuoteInToken does not work if STBC is != 18 decimals
Severity	Low
Description	The _getQuoteInToken function is responsible for returning how much of the RWA asset is equivalent to the provided USD amount. This is done by fetching the USD price of the RWA and the RWA decimals and then dividing the STBC (USD amount) by the price: return amount * 10 ** decimals / priceInUSD; Which could be something like: 100e18 * 1e18 / 0.9e18

	resulting in 111e18 RWA for 100 USD if the RWA is worth 0.9USD This calculation will however not work if the STBC is in != 18 decimals, as example for 6 decimals: 100e6 * 1e18 / 0.9e18 resulting in 111e6 RWA for 100 USD. *This issue is only rated as low severity since STBC's decimals are always 18 in the current architecture.
Recommendations	If in the future there is a modification of the protocol planned which allows adding STBC with != 18 decimals, the value here should be normalized to 18 decimals before the calculation.
Comments / Resolution	Resolved, this logic has been refactored.

Issue	**Coefficient calculation will be flawed if same STBC is used for more than 2 buckets**
Severity	Informational
Description	The coefficient calculation is using the totalSupply of the STBC: totalUsdSupply = IERC20(_buckets.getRWABucketStablecoin(BUCKET_RWA_DAO_COLLATERAL)).totalSupply(); If the same STBC is used for more than 1 bucket, this will result in a deflated coefficient, ultimately resulting in users becoming less tokens than expected during the redeem.

	This issue is only rated as informational since for V0 there will never be more than 1 RWA bucket.
Recommendations	For V1 consider explicitly checking that the same STBC is not assigned to more than one bucket.
Comments / Resolution	Acknowledged.

Issue	Certain variables can be made immutable
Severity	Informational
Description	Variables which are defined within the constructor and never changed afterwards can be marked as immutable, this will include these variables directly into the contracts bytecode, which will save gas whenever these variables are accessed. L 102 - 117 IRegistryAccess private _registryAccess; ... ITokenMapping private _tokenMapping;
Recommendations	Consider marking the aforementioned variables as immutable.
Comments / Resolution	Resolved.

Issue	Typographical Issues
Severity	Informational
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below: L 17: import {IStbc} from "src/interfaces/token/IStbc.sol"; This import is duplicated. L 125: event Swap(address indexed tokenSwapped, uint256 amount, uint256 lusRewards, uint256 amountInUSD); The lusRewards parameter should be usDaoRewards. L 129: event SwapRWAForStbcMintEngine(address indexed rwaProvider, address indexed stableOwner, address rwaToken, uint256 amount, uint256 priceInUSD, uint256 usdaoInstant, uint256 userUsDAO); The usdaoInstant parameter should be LusDaoInstant. L 155:

	event TransferRewardsToTreasury(address indexed from, uint256 lusRewards); This event is unused. L 477: function _transferAndMint(address tokenToSwap, uint256 amount) This function is unused.
Recommendations	Consider fixing the aforementioned issues.
Comments / Resolution	Resolved.

TokenMapping

The TokenMapping contract allows the usualTech role to assign RWAs to a specific STBC via a simple key-value mapping. Additionally it allows the usualTech role to declare all RWAs for one specific STBC, such that one can fetch directly all corresponding RWAs from one specific STBC.

These additions are reversible using the corresponding “remove” functions.

These settings are then used by the DaoCollateral contract on several occasions.

Threat Modeling:

The main risk for this contract is due to incorrect configuration

Issue	Governance Privilege: Potential DoS
Severity	Governance
Description	The core of the DaoCollateral functionality relies on the correctness of the settings within this contract. If set incorrectly, this can break core functionalities within the DaoCollateral contract.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue	Removal of RWA from STBC via removeStbcToRwa will result in DoS upon _calculateCoefAndRemainingToMint, permanently preventing the CBR mode
Severity	**High**

Description	The contract allows for two important configurations:
	
	a) setRwaToStbc: Determines the corresponding STBC address for a RWA, is used within the DaoCollateral on multiple occasions whenever a RWA address is provided.
	
	b) setStbcToRwa: Sets corresponding RWAs to a STBC. This function allows for multiple RWAs to be assigned to the same STBC and keeps track of the amount via incrementing the following mapping:
	
	++_stbcToRwaLastId[stbc]
	
	Now, the latter function corresponds to the getAllRwaFromStbc function, which is invoked upon the coef calculation and simply returns all RWAs for the STBC.
	
	The problem now lies within the fact that the corresponding RWAs can also be removed from a STBC again via the removeStbcFromRwa function. The crux with this function is that the _stbcToRwaLastId[stbc] variable is only decreased if the last RWA is removed:
	
	// if it is the last rwa, we can decrease the last id
	if (_stbcToRwaLastId[stbc] == rwaIndex) {
	_stbcToRwaLastId[stbc] -= 1;
	}
	
	Which means if we do not remove the last RWA, it is simply set to address zero but still existent within the loop.

	This is a problem since the _calculateCoefAndRemainingToMint function inherently fetches the price of all RWAs in the array. In the scenario that an RWA is address(0), the getPrice call will revert, which then results in a permanent DoS of the _calculateCoefAndRemainingToMint function, effectively preventing the activation of the CBR mode.
Recommendations	Consider using the swap and pop mechanism to effectively remove the RWA from the array. It goes without saying that also the _stbcToRwaLastId[stbc] variable must be decreased accordingly during every removal. Optionally another fix would be to not include sparse elements within the getAllRwaFromStbc function.
Comments / Resolution	Not resolved, the same issue exists upon removal. Additionally due to the newly implemented upper limit, this introduces another issue: If RWAs which are not the last index are removed, this will not decrease isUsUSDCollateral, resulting in a DoS of new additions at some point.

Issue	Same RWA token can be added multiple times to a STBC
Severity	Low
Description	Currently, within the setStbcToRwa function, there is no check that a linkage already exists. This will result in miscalculations within the coef calculation.
Recommendations	Consider if it makes sense to implement such a check. Essentially this would include a loop over all elements, checking if such a combination already exists. Alternatively, we are fine with this issue being acknowledged while being careful when adding linkages.

Comments / Resolution	Resolved.

Issue	Loop within getAllRwaFromStbc function can run out of gas
Severity	Low
Description	The getAllRwaFromStbc function loops over all indexes for specific STBC to RWA linkage. In the scenario where the array becomes unreasonably large, this loop will consume excessive gas which can result in a function revert due to the “out of gas” issue. This issue has only been marked as low since no gas-consuming execution happens within the loop and in practice there won’t be unreasonably amounts of RWAs added to a STBC
Recommendations	Consider staying reasonable wich RWA to STBC linkages. Additionally one can include a cap on the index increase. However, this means the removal must correctly decrease the index.
Comments / Resolution	Resolved.

Conclusion

Our comprehensive audit of the protocol has revealed a significant number of issues, particularly those classified as high and medium severity. The presence of numerous high and medium severity issues suggests that the development cycle of the protocol is not yet complete. This is further evidenced by certain areas in the codebase that appear to be underdeveloped or partially implemented. A notable absence of edge-case testing has been observed. Edge-case testing is crucial for ensuring the robustness of the protocol under a wide range of conditions, especially in scenarios that are not immediately obvious or are less likely to occur. Addressing the identified bugs and vulnerabilities will likely require more than simple patches. A slight refactoring of the codebase may be necessary to ensure that fixes are comprehensive and do not introduce new issues.

Given the large number of issues discovered, addressing them all will require substantial effort and resources. A standard resolution round, which typically allocates between 2.5% to 5% of the initial audit allocation, will not be sufficient to fully resolve all the identified problems, especially will this not fully cover the observation of new issues which might have been introduced during bug-fixes, from experience about 25% of bug-fixes introduce new issues. Additionally related to the overall topic but not related to the current state of the contract, the team indicated that they have already worked on a few small out-of-scope changes which do not necessarily have a bug as root-cause. This will further manipulate the resolution codebase.

We recommend the following:

1. A thorough cleanup of the codebase to eliminate existing issues and improve overall code quality.

2. All high and medium severity issues, as well as lower severity ones, should be systematically addressed and resolved

3. Where necessary, consider adjusting the underlying logic of the protocol to enhance security and efficiency.

4. Develop and implement a fully-loaded testing suite, encompassing a wide range of scenarios including edge cases. This is vital for ensuring the protocol's reliability and security.

5. Once these steps have been completed, we strongly advise re-submitting the finalized contract for a comprehensive re-audit. This will provide an opportunity to

validate the fixes and ensure that no new issues have been introduced during the refactoring process. The measures suggested above are aimed at elevating the protocol to a level of maturity and reliability that is essential for its success and user trust, especially in the world of smart contracts.

Specifically since the next audit is planned by Spearbit, we recommend to clean up the codebase before and conduct a second round by BailSec, with the goal to allow Spearbit's researchers to focus on deep logical issues and advanced exploit methodologies. The application of advanced exploit methodologies will only work on codebases which are *almost* completely free of most obvious bugs. A different approach is applied for this specific purpose, which was not possible to apply during this audit round due to the sheer amount of issues. In such a scenario that there is no additional re-audit, the codebase will still have bugs due to incorrect fixes, adjusted logic and other issues. If this codebase is provided to Spearbit for the final audit, the researchers cannot to 100% focus on advanced exploit methodologies, such as exploiting complex function flows, input scenario exploits or complex economical exploits, to just name a few.

The other possibility would of course be to do the standard resolution round with the allocated resources, this will mostly cover the 1:1 fixes but cannot sufficiently secure larger fixes and logical changes. Of course it goes without saying that there will be quite a substantial amount of bugs left after such a standard resolution round. At the end of the day, the client decides on which route to go, we can just recommend best practices based on our large expertise as auditors.

Post-Audit

Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	16			
Medium	6			
Low	16			
Informational	14			
Governance	1			
Total	53			

Post-Audit Review

The UsUSD team's updated commit in response to our audit that identified a considerable number of issues—27 high, 21 medium, and 48 low severity—underscores the complexity and the critical need for rigorous security measures within their smart contract codebase.

Given the substantial volume of vulnerabilities discovered, a simplistic resolution round was deemed infeasible, necessitating a more structured and focused approach to validate and rectify the identified issues.

Therefore, a new review round was commenced. The approach outlined for this review encompasses two primary goals within a constrained 14-day period.

First, it aims to validate all the fixes applied in response to the audit findings, ensuring that each corrective measure adequately addresses the vulnerability it was intended to fix. Second, it involves conducting a thorough re-audit of the entire codebase from scratch. This dual-focus methodology is designed not only to verify the effectiveness of the fixes but also to identify any additional vulnerabilities that may have been introduced during the process of implementing these fixes or that were overlooked in the initial audit.

However, it's crucial to highlight the significant limitation posed by the 14-day review period. Given the complexity of the codebase and the sheer volume of issues initially identified, a comprehensive re-audit within such a restricted time frame is inherently constrained. Ideally, a more extended review period, such as four weeks, would provide a more suitable window to conduct an exhaustive examination of the codebase, ensuring a more thorough validation of fixes and identification of potential vulnerabilities. This truncated review period implies that the re-audit will inevitably leave significant portions of the codebase unexamined, with a high likelihood that some high-severity issues will remain unaddressed, potentially even a large amount of issues.

The reason that only a 14 day period was allocated is the fixed audit start from Spearbit, on 4th march. This review is conducted by one single senior auditor from 19th february to 3rd march. This will not be a full audit and is also not intended to serve as a full audit in any scenario.

The following commit was provided:

https://github.com/usual-dao/pegasus/tree/9e42b9590bf62373d066245ad5a35f8177d1f375/packages/solidity/src

In the following, we will only list issues which are new. Issues which have been not resolved will not be added again.

*To simplify PoC's, we will use USDT as a minter asset. We will use 1e6 decimals for RWA and USDT, however, we still may raise issues for the case where the minter asset has 18 decimals, just to increase awareness for the future.

*Note that this audit will mostly not include written PoC's for simple time-management reasons. There are only 14 days allocated to this review and by saving PoC's we can allocate more time to the actual review.

Note:

Since now 14 days have passed, we determine the coverage reached to about 65%. The main missing steps are line-by-line checks for most contracts. While we have covered mainly logical flows and cross-contract-interactions, low-hanging fruit might still be present which is only uncovered once every single line is checked.

Bucket

BucketDistribution.sol

Issue	Governance Privilege: BucketCreator can steal assets
Severity	Governance
Description	The createBucket function inherently grants the provided distributor the right to spend tokens. This can be abused by a malicious BucketCreator to drain all tokens in the contract. Since the protocol has a layered governance structure with different roles and (presumably) different corresponding trust-levels, this can become a critical issue.
Recommendations	Consider if this can become an issue - depending on how trusted addresses with this role are. If significant, consider only allowing the DAO address to create new buckets.
Comments / Resolution	

Issue	Implicit assumption that bucket = BUCKET_DAO_USDAO_TREASURY handles usDAO can lead to loss of funds.
Severity	Low
Description	Whenever the _mintUsDAO function is invoked, usDAO is minted to address(this) and the balance of the BUCKET_DAO_USDAO_TREASURY is increased. That logic has the implicit assumption that the BUCKET_DAO_USDAO_TREASURY is in fact responsible for usDAO.

	If that is not the case, these tokens are stuck.
Recommendations	Consider setting this bucket upon deployment.
Comments / Resolution	

BondDistributor.sol

Line by line check is missing for this contract.

No new issues found.

LpDistributor.sol

Line by line check is missing for this contract.

Issue	Incorrect receiverPool can result in permanently locked funds
Severity	Low
Description	The distribute function first transfers usDAO from the BucketDistribution contract in, then locks it and calls the deposit_reward_token function on the receiverPool. However, if the receiverPool does not execute a transferFrom, the LusDAO would effectively be locked in the LpDistributor contract.
Recommendations	Consider implementing a before-after balance check to ensure that the profitBalance was in fact transferred out: beforeBalance = lusDao.balanceOf(address(this)); IGauge(receiverPool).deposit_reward_token(address(lusdao), profitBalance);

	afterBalance = lusDao.balanceOf(address(this)); assert(beforeBalance - afterBalance == profitBalance);
Comments / Resolution	

ManualDistributor.sol

Line by line check is missing for this contract.

Factory

LsausUSDFactory.sol

Line by line check is missing for this contract.

Issue	_sendReward is invoked with block.number instead of bondTime which will result in incorrect Bond funding
Severity	High
Description	Whenever a new bond is deployed, the bondTime variable is determined, which is usually the time when the bond starts giving out rewards. The _sendReward function is however invoked with the current block.number, which means that rewards are not calculated from bondStart on but rather from the current timestamp. This issue is then further moved on to the LsausUSD token, since the rewards are distributed based from this block.number, however, the close function incorporates the bondStart for the refund. Even though, with the bug being consistent in the LsausUSD token, which means that rewards are covered, rewards will be permanently lost because the close function uses the bondStart parameter to fetch the total duration (instead of the initial block.timestamp when the bond was deployed).
Recommendations	Consider correctly incorporating the bondStart for the _sendRewards function and the rewardPerBlock calculation, as well as starting reward distribution only once the bond has started.
Comments / Resolution	

Issue	Exact same Bond cannot be used twice due to safeguards
Severity	Informational
Description	The createLsausUsd function allows the DAO address to do the following: 1. Craft a new bucket (bytes32) 2. Deploy a new (unique) LsausUsd contract 3. Deploy a corresponding distributor 4. Create a new bucket This function allows for multiple parameters such as the fee, supply, etc. Additionally, the name and symbol can be determined. Amongst all these parameters, the bucket variable is crafted which is then later created. Due to the createBucket function only being callable with not yet existent buckets, there is no possibility in using the same name/symbol twice.
Recommendations	While we do not assume that such a practice is planned, it is still important to informationally raise this issue. This can be safely acknowledged.
Comments / Resolution	

MintEngine (special careful)

A notable change to the architecture has been implemented in an effort to meet the modular USD value approach for existing offers. On a high level overview it can be simply described as follows: Upon offer creation (RWA and Minter), the nominal token amounts for the offer creation are pushed into the orderbooks. Whenever a new offer is now created with an existing counter offer, the USD value for both offers is then fetched and matched with each other to ensure fair and correct order management even with fluctuating token prices. Here is how this is done for the specific scenarios:

1. RWA offer creation:

1.1 RWA offer is created with no existing minter offer

- The RWA offer is simply pushed with its nominal value in the orderbook, which then waits until it is fetched.

1.2 RWA offer is created with existing minter offer, both USD values are exactly the same

- Both offers are fully matched and removed, there are no leftover offers.

1.3 RWA offer is created with existing minter offer, the minter is worth more than the RWA offer (in USD terms)

- Both offers are matched but the leftover amount in the minter offer is still existing as valid order, with its remaining amount.

1.4 RWA offer is created with existing minter offer, the minter offer is worth less than the RWA offer (in USD terms)

- Both offers are matched, the minter offer is removed completely since it is fully consumed
- The leftover amount of the RWA offer is pushed into the CP orderbook with its nominal leftover amount, waiting to be picked up by future minter offers

2. Minter offer creation:

2.1 Minter offer is created with no existing RWA offer, the nominal USDT amount is pushed into the MinterPool, waiting to be picked up by the RWA offer.

2.2 Minter offer is created with existing RWA offer, both USD values are exactly the same

- Both offers are fully matched and removed, there are no leftover offers.

2.3 Minter offer is created with existing RWA offers, the RWA offer is worth more than the minter offer (in USD terms)

- Both offers are matched but the leftover amount in the CP orderbook is still existing as valid order, with its remaining amount.

2.4. Minter offer is created with existing RWA offer, the RWA offer is worth less than the minter offer (in USD terms)

- Both offers are matched, the RWA offer is completely removed since it is fully consumed.
- The leftover amount of the Minter offer is pushed into the MinterPool with its nominal leftover amount, waiting to be picked up by future RWA offers. The leftover order cannot be canceled.

Additionally, all cancellations will happen with the nominal token amount.

Issue	Unintentional creation of leftover offer due to price fluctuations will result in slight disadvantage for users
Severity	Low
Description	The MintEngine has been refactored in an effort to work with fluctuating prices. This includes that all existing offers will be converted to USD and matched with the provided USD value. Whenever a user therefore creates an order, it is matched with the existing counter offer in USD denomination. Let's now assume the following: RWA price = 1 USDT price = 1

	1. Alice observes 1000 USDT in the MinterPool 2. Alice creates an offer with 1000 RWA 3. The transaction is stuck in the mempool, the USDT price decreases by 1% and the RWA price increases by 1% 4. The following USD values are now existent: Alice's offer: 1010 USD MinterPool: 990 5. Alice's offer is now matched with the funds in the minter pool, while Alice assumes that her offer is fully covered, it is not. Since created offers in the MinterPool cannot be canceled anymore, Alice will end up with temporarily stuck funds. This issue is also present with frontrunning.
Recommendations	Consider incorporating a slippage mechanism.
Comments / Resolution	

MintEngineOrder.sol

Issue	Deposit within createOfferMinter is executed even if there are existing RWA offers, resulting in users gaining unlawful position within the MintEngineMinterPool
Severity	**High**
Description	The createOfferMinter function allows a user to create a minter offer (USDT). Mainly, there are three possible scenarios: 1) Existing RWA offers for USDT in the orderbook and the offers can cover the USDT offer 2) Existing RWA offers for USDT in the orderbook and the offers can NOT cover the USDT offer 3) Non-existing RWA offers for USDT in the orderbook For scenario 2), a deposit is made to the MinterPool and the function returns. For scenario 3), a deposit with the leftover amount should be made. In the previous audit round, we already mentioned that a duplicate deposit is happening for the full offer and for the leftover amount. The duplicate deposit was removed, however, for scenario 1), the deposit is still executed, which creates a valid minter offer position for the user while in fact the provided USDT was already exchanged. The result is that the USDT provider received the STBC but additionally has a full minter offer.
Recommendations	Consider only executing a full deposit when the orderbook is empty. Ie. within the if clause:

	if (mintEngineCpOrderbook.isCpOrderbookEmpty(token)) {
	
	mintEngineMinterPool.deposit(msg.sender, token, amount);
	return;
	
	}

Comments / Resolution	

Issue	Lack of validation for setMinimumRwa
Severity	Low
Description	There is no proper lower or upper validation for the said function. This can result in governance accidentally or maliciously setting unreasonable thresholds, which will then negatively impact the business logic.
Recommendations	Consider implementing a proper validation for this variable.
Comments / Resolution	

Issue	Inconsistency in function validation between createOfferMinter and createOfferCpToken
Severity	Low
Description	The createOfferMinter function only allows the amount parameter to be uint96, the createOfferCpToken however does not expose this check. We are of the opinion that both functions should validate the amount parameter consistently. Moreover, the amount should be validated to be non-zero. For both functions.

Recommendations	Consider incorporate the following check towards the createOfferCpToken function:
	
	if (amount > type(uint96).max) {
	  revert AmountTooBig();
	}

Comments / Resolution	

Issue	Lack of event for WL removal
Severity	Informational
Description	The contract exposes an AddWhitelistToken event, however, for the removal there is no corresponding event. This can result in issues for third-parties that rely on correct event emission.
Recommendations	Consider implementing an event for the WL removal.
Comments / Resolution	

MintEngineCpOrderBook.sol

Issue	Advanced exploit allows for breaking invariant “only one offer for RWA / Minter should exist at a time”, resulting in skipped IDs
Severity	High
Description	The orderMatching mechanism loops over all offers until numOrderTook becomes 10, this safeguard is implemented in an effort to prevent any DoS attacks. It has to be mentioned that this safeguard is rather from a theoretical perspective, since RWA offers can only be created with a minimum value (10 000 USD) and therefore it is (practically) impossible that a malicious user creates 200+ orders with 1 wei to DoS the loop. (The cancellation of thees incurs a fee for the creator) Unfortunately, we uncovered an attack to use this safeguard against the protocol and break one of the most important invariants: “There should only be either an active minter offer or an active RWA offer at the same time”

	Here's how this exploit unfolds: 1. Alice creates 11 RWA offers with each being worth 10.000 USD 2. Alice creates a minter offer with 10.500 USD, since there are existing RWA offers, this will undergo the matching mechanism. 3. The matching mechanism loops over all offers, matching 0-9 and will then stop. 4. This will only match 10.000 USD, leaving 500 outstanding. 5. For the leftover, there is now a deposit of 500 USD The contract is now in a state where there is an existing RWA and Minter offer. Given that this scenario not only happens intentionally but also during the normal business logic, there is specifically one issue which could occur afterwards: 1. RWA and MinterOffers are both existent 2. Bob creates a MinterOffer which is partially matched with the RWA offer, there is a leftover amount. 3. This leftover amount is used to create a new MinterOffer. This will increase currentIdPerToken 4. Due to the currentIdPerToken increase and the fact that the first existing MinterOffer has not been fully matched, it is clear that there is absolutely no possibility to allocate the necessary rewards to the first offer, since the new matching mechanism will add rewards to the currentIdPerToken, effectively preventing the first MinterOffer to get the corresponding STBC.
Recommendations	After some considerations, given the minimum amount for a RWA offer (10 000 USDT), we observed that a DoS possibility due to the normal business logic is more or less impossible.

	Therefore, we recommend simply removing this safeguard, we further advised the team to further construct testing scenarios with large loops >100, >200 and compare these with Ethereums block gas limit of 30 000 000.
Comments / Resolution	

Issue	Usage of wrong denomination for arithmetic operation within orderMatching function will break the contract
Severity	**High**
Description	The orderMatching function is responsible for matching a newly created USDT offer with existing RWA offers. It does that by looping over all RWA offers and either fully or partially taking them (take). If a RWA offer is not sufficient to match the USDT offer, the remaining orderAmount is calculated as follows: uint256 orderAmount = _getAmountInToken(rwaToken, amount - amountMatched) The problem is that amountMatched is denominated in nominal RWA tokens, while the amount is denominated in the USD value, therefore, the operation: uint256 orderAmount = _getAmountInToken(rwaToken, amount - amountMatched) Executes an arithmetic operation using two different denominations: USD value - RWA amount This will result in a flawed calculation, falsifying the rest of the function logic and potentially resulting in a revert due to underflow.

	Essentially it will just break the matching mechanism completely.
Recommendations	Consider converting the amountMatched to the corresponding USD value before this calculation.
Comments / Resolution	

Issue	Cancellation does not work if RWA provider is blacklisted from RWA
Severity	High
Description	The cancelOffer function allows the DAO address to force cancel an offer. This freedom is necessary in such a scenario where the RWA provider is blacklisted for USDT, as this would prevent the outstanding order from being fulfilled. (The order matching would revert if the USDT is transferred to the RWA provider) However, an additional problem was observed: If the RWA provider is blacklisted for the RWA as well (after he has already created the offer of course). In such a scenario, the cancelOffer function will revert because the RWA is transferred back to the provider (which is blacklisted). This would essentially break the MintEngine permanently.
Recommendations	Consider incorporating an additional cancelOfferGovernance function, which provides an address parameter to transfer the RWA to a different address, effectively circumventing the blacklisting (but also take the tokens from the RWA provider).
Comments / Resolution	

Issue	Incorrect return value within orderMatching function breaks matchCpOffer function
Severity	High

Description	The matchCpOffer function within the MintEngineMatching contract follows these steps:
	
	a) Convert the provided USDT amount to the USD value
	
	b) Invoke the **orderMatching** function within the MintEngineCpOrderbook contract
	
	c) Invoke the executeOrderCp function within the MintEngineExecutor contract
	
	d) Eventually create a minter offer if the orders within the MintEngineCpOrderbook were not sufficient to cover the USDT offer
	
	The part which is of interest for us is d), which depends on the return value of the orderMatching function. This brings us also to the issue which is described follows:
	
	The orderMatching function loops over all existing RWA offers and has two different return scenarios:
	
	a) Return whenever the orderAmount is matched:
	
	if (matched == orderAmount) {
	
	return (**amount**, transferInfos);
	
	}

	This will return the amount = USD-Value
	
	b) Return whenever all orders have been consumed/the loop is finalized (this is the case when the orders cannot cover the incoming amount:
	
	return (**amountMatched**, transferInfos);
	
	As one can see, this will return **amountMatched**.
	The problem is that this is the nominal RWA amount, which was matched:
	
	(Queue.Offer memory offer, uint256 matched) = _offer[token].take(orderAmount);
	
	amountMatched += matched;
	
	Therefore, we have two different return values in two different denominations, one time as the USD value and the second time as nominal RWA value.
	
	The matchCpOffer function executes the following condition check:
	
	if (amountMatched == amount)
	
	where amountMatched is the return value discussed before.
	
	Therefore, this condition check is comparison the amount to the return value (amount is USD value; amountMatched is RWA amount)

Recommendations	Consider returning the USD value of the matched amount to ensure a correct condition check.
Comments / Resolution	

Issue	Incorrect parameters for the swapRwaForStbcMintEngineCp call during order cancellation will result in incorrect fee handling
Severity	Medium
Description	Users have the right to cancel their RWA offers in the orderbook via cancelOrder. This will apply a nominal fee of 0.001%: uint256 halfBasisPoint = (value * 1) / SCALAR_HUNDRED_KWEI This fee is taken from the RWA amount the user has offered and is then used upon the swapRwaForStbcMintEngineCp call. The problem lies within the fact that the swapRwaForStbcMintEngineCp function expects the amount parameter to be the USD value of the RWA amount, however, the **nominal fee amount** is provided here: _daoCollateral.swapRwaForStbcMintEngineCp(bucket, bucket, token, fees) This means that the transferFrom attempts to transfer in an invalid amount, since it expects the USD value to be the amount parameter, while it is the **nominal fee amount**: amount * SCALAR_ONE_SIX / _ORACLE_USUAL.getPrice(rwaToken)
Recommendations	Consider using the USD amount of the corresponding fee for the swapRwaForStbcMintEngineCp call. This can be simply done by converting the fee to the USD amount beforehand.
Comments / Resolution	

Issue	View function getOfferFromId displays incorrect offer
Severity	Low
Description	The function getOfferFromId is a view-only function for users and third parties, it fetches the corresponding offer as follows: return _offer[token].getOfferFromId(offerId - 1); This is incorrect as offerId is in fact the corresponding index.
Recommendations	Consider removing -1.
Comments / Resolution	

Issue	Order cancellation can result in theoretical DoS of matching mechanism
Severity	Low
Description	Whenever an offer is canceled, it is being set to the CANCELED state within the Queue contract. Upon the order matching mechanism, a loop is executed over all orders and if the order is canceled, it is deleted and the _begin index increased, such that the next order can be fetched. Theoretically a user could create thousands of orders and cancel them again, maliciously increasing the list queue. This could then theoretically result in the orderMatching mechanism to run out of gas because the DoS safeguard would not catch it. However, there are several facts that downgrade this issue to informational, as it is practically impossible. A user would need to create and cancel thousands of orders for 10.000 USD and each

	cancellation occurs at a fee. Moreover, due to the low gas consumption for canceled orders, this would literally need thousands of orders to play out. Initially, we rated this issue as informational, however, due to the low cancellation fee of 0.001%, this issue might still happen in practice.
Recommendations	Consider re-thinking the low fee of 0.001%, comments indicate that it should be 0.005%. However, even this might make the attack practically possible.
Comments / Resolution	

Issue	Comment about cancellation fee is incorrect
Severity	Low
Description	BPS stands for Basis Points, where one basis point is equal to 0.01% (one hundredth of a percent). In the function, SCALAR_HUNDRED_KWEI is 100,000 for easier understanding: (value × 1) / SCALAR_HUNDRED_KWEI → (value × 1) / 100000 This is equivalent to dividing the original value by 100,000, which is essentially calculating 0.001% of the input value. Since 0.001% = 0.1 BPS, the function calculates and subtracts a 0.1 BPS fee. Therefore, the comment: "/// @dev This internal utility function calculates the value minus fees based on a 0.5 basis point fee rate."

	Is incorrect.
Recommendations	Consider fixing either the comment or the code itself.
Comments / Resolution	

Issue	**_scaleAmountByDecimals is unused**
Severity	Informational
Description	The aforementioned function was used in the previous function and is not used anymore. The existence of unused codes will confuse third-party reviewers.
Recommendations	Consider removing it.
Comments / Resolution	

MintEngineMatching.sol

Line by line check is missing for this contract.

Issue	Leftover offer creation for MinterPool has no correct corresponding deposit which makes the offer invalid
Severity	**High**

Description	The matchCpOffer function invokes the orderMatching function, which loops over all existing RWA offers in the MintEngineCpOrderbook:
	
	mintEngineCpOrderbook.orderMatching(tokenToExchange, amount);
	
	afterwards, it then eventually creates a virtual position if the existing RWA offers are insufficient to cover the created USDT offer:
	
	mintEngineMinterPool.createOffer(tokenToExchange, amount - amountMatched);
	
	The problem is that this new offer creation has no corresponding deposit, which means that even if this virtual position is created and fulfilled, the increase of the claimablePerOffer variable will have no effect, as the corresponding user can not claim it.
	
	This issue is only valid once the lose deposit within the createOfferMinter function in the MintEngineOrder contract is fixed (the deposit should only happen if there is no existing RWA offer, otherwise the code is broken anyways)

Recommendations	Consider triggering the deposit function with the correct leftover amount before the offer creation. It is important to trigger the deposit before the offer creation to match the deposit with the currentIdPerToken

Comments / Resolution	

Issue	Leftover offer creation within matchMinterOffer is using incorrect denominations for arithmetic operation
Severity	High
Description	The matchMinterOffer function is responsible for invoking the poolMatching function within the MintEngineMinterPool, executing the offer and eventually creating a leftover offer if the USDT amount in the MinterPool is insufficient to cover the RWA offer. The leftover creation is done as follows: mintEngineCpOrderbook.createOffer(caller, tokenToExchange, amount - amountMatched, tokenPreferred); The most important invariant for offer creations is that all offers are created using the nominal token amount. This is necessary to ensure it can be later upon matching converted to the corresponding USD value. This invariant is violated because of the following facts: amount = USD value of RWA amount amountMatched = matched USDT amount for amount Therefore, the result of the arithmetic operation is not only incorrect, it can also revert due to an underflow if USDT is below 1, since that would mean amountMatched > amount. Additionally, this can result in a state where an illegal offer is created while there are still existing funds in the MinterPool. This will completely break the contract.
Recommendations	Consider using the USDT denomination for the arithmetic operation.

Comments / Resolution	

Issue	Leftover offer creation within matchCpOffer is done using the USD value instead of nominal USDT value
Severity	High
Description	The matchCpOffer function creates an offer in the scenario where the corresponding RWA offers were insufficient to cover the provided USDT value: mintEngineMinterPool.createOffer(tokenToExchange, amount - amountMatched) This offer is created with the USD value instead of the nominal USDT value. This breaks an important invariant of the protocol
Recommendations	Consider converting the USD value delta: amount - amountMatched To the corresponding USDT amount. It has to be noted that the orderMatching function itself also has an incorrect return value for one condition, this would need to be fixed as well.
Comments / Resolution	

Issue	Loose condition check within matchMinterOffer does not safeguard against underflows
Severity	**High**

Description	Within the matchMinterOffer, the following check is existent:
	
	if (amountMatched != amount)
	
	While (once fixed), the function flow should technically prevent amountMatched > amount, we are of the opinion that a strict check is better than a loose check.
	
	This issue has been increased to high after it was observed that the following arithmetic operation is using two different denominations.

Recommendations	Consider changing the condition to this:
	
	if (amount > amountMatched) {
	..
	}

Comments / Resolution	

Issue	matchCpOffer will not work if tokenToExchange has 18 decimals
Severity	Low
Description	The matchCpOffer function is responsible for forwarding the amountToMatch to the orderbook mechanism and then execute an order while eventually creating an offer if the orderbook cannot cover the offer. Before the call to the orderbook, the amountToMatch is converted to the corresponding USD value. This will work for tokens with 6 decimals but not for tokens with 18 decimals due to the implementation of the getQuote function which divides the result by 1e6. This issue is only rated as low severity since it is only planned to use USDT and USDC which both have 6 decimals.
Recommendations	Consider rewriting the getQuote function such that it divides not by 1e6 but by the actual decimals of the token: price * amount / token.decimals()
Comments / Resolution	

Issue	Lose condition check within matchCpOffer can result in underflow
Severity	Low

Description	The matchCpOffer function matches the incoming USDT offer with existing RWA offers. If the matching was insufficient and there were insufficient RWAs to match the USDT amount, a leftover offer with the remaining USDT amount is created.
	
	Such a condition is checked with an early return:
	
	if (amountMatched == amount) {
	return;
	
	}
	
	In other words, if not the full amount was matched, the leftover offer is created. In itself and also in relation to the overall business logic this condition check is safe and sound. However, we are still of the opinion an explicit condition check is more secure as this would prevent from underflows.

Recommendations	Consider adjusting the condition check as follows:
	
	if (amount > amountMatched) {
	
	mintEngineMinterPool.createOffer(tokenToExchange, amount - amountMatched);
	
	}

Comments / Resolution	

MintEngineExecutor.sol

Line by line check is missing for this contract.

Issue	executeOrderMinter uses nominal USDT amount for swap which results in incorrect amountMatched parameter for swapRwaForStbcMintEngineMinter call
Severity	High
Description	The executeOrderMinter function is invoked upon a RWA offer creation and responsible for the following tasks: a) Exchanging RWA to STBC via swapRwaForStbcMintEngineMinter b) Transferring USDT to the initiator (RWA provider) c) Adding STBC rewards for the corresponding offerID First of all, we need to understand that the previous matching mechanism is using the USD value of the provided RWA amount to match with USDT in the minter pool: uint256 amount = rwaOracle.getQuote(tokenToExchange, amountToMatch) -> the nominal amountToMatch amount is here converted to the corresponding USD value (uint256 amountMatched, bytes[] memory transferInfos) = mintEngineMinterPool.poolMatching(tokenPreferred, amount) -> poolMatching directly invokes _matchOrderbook with the USD value function _matchOrderbook(address token, uint256 amount, bytes[] memory transferInfos) -> until here, amount is the nominal USD value. amount = _oracle.getQuote(token, amount) *-> this is already flawed and will return an incorrect value, the amount parameter is already the USD value of the RWA asset*

-> this should then convert the USD value back to the corresponding USDT amount. (we have raised a separate issue for this)

(Queue.Offer memory offer, uint256 amountMatched) = _offer[token].take(amount)
-> here we assume the above mentioned bug is fixed and amount is the real USDT amount

transferInfos = _appendToBytesArray(transferInfos, abi.encode(token, uint96(amountMatched), address(this), uint96(offerId)))
-> we include the matched USDT amount in the transferInfos array

(address token, uint96 amount, address owner, uint96 offerId) = abi.decode(transferInfos[i], (address, uint96, address, uint96))
-> we decode the matched USDT amount again

Now the impact:

If for example 100 RWA tokens are worth 110 USD, this means 110 USDT are matched (amount = 110e6). One can simply follow the call-path above to find this out.

and 110e6 are attached to the transferInfo array.

The problem is that, while provided 100 RWA, the function approves and uses the amount parameter, which is, as explained 110e6:

IERC20(address(rwaToken)).approve(address(_daoCollateral), amount)

_daoCollateral.swapRwaForStbcMintEngineMinter(caller, owner, rwaToken, amountStable, amount)

While the amount parameter is the matched USDT amount, which can be, as already explained, 110 instead of 100. But the swapRwaForStbcMintEngineMinter expects the nominal RWA amount, instead the matched USDT amount.

	This will then further result in downstream issues. such as transferring 110 RWA tokens in, while the offer created only provided 100: IERC20(address(rwaToken)).safeTransferFrom(msg.sender, address(_BUCKETS), amount) If the 100 RWA tokens would be worth only 90 USD, this would then match 90 USDT and approve and swap 90 RWA.
Recommendations	Consider converting the matched USDT amount (amount) to the corresponding nominal RWA amount.
Comments / Resolution	

Issue	Important balancePerToken implementation lacks decrease
Severity	High
Description	Whenever a user creates a RWA offer with existing minter offers, the balancePerToken is increased by the RWA amount: mintEngineExecutor.depositExecutor(msg.sender, token, amount) If there are existing funds in the minter pool, the RWA offer is matched with these and the swapRwaForStbcMintEngineMinter is invoked The USDT amount is correctly decreased: balancePerToken[token] -= amount The RWA amount is however not.

	—————————————————————————————
	Respectively for the creation of a minter offer with existing RWA offers, the balancePerToken is increased by the USDT amount:
	
	mintEngineExecutor.depositExecutor(msg.sender, token, amount)
	
	Afterwards, the minter offer is matched with RWA offers in the MintEngineCpOrderbook and the order is executed.
	
	During the order execution, the balancePerToken for the RWA token is decreased:
	
	balancePerToken[token] -= amount * SCALAR_ONE_SIX / rwaPrice
	
	The USDT amount is however not.
	
	Specifically in the second scenario where USDT is not decreased, this will then still mark minter offers as existing, while they eventually are not:
	
	function isMinterPoolEmpty(address token) external view returns (bool) {
	
	return getBalancePerToken(token) == 0;
	
	}
	
	If now, new RWA offers are created, an attempt is made to match these with minter offers, however, there are no corresponding offers / tokens in the contract and the call will then revert.

Recommendations	Consider decreasing the RWA amount and USDT amount as well.
Comments / Resolution	

Issue	executeOrderCp uses RWA USD-Value approval instead of nominal RWA value
Severity	**High**
Description	The executeOrderCp function is invoked upon successful order matching whenever a USDT offer is created and is responsible for the following tasks: a) Decoding the transferInfo array b) Approving the RWA to the DaoCollateral contract c) Invoke the swapRwaForStbcMintEngineCp function to transfer RWA in and mint the STBC to the USDT provider e) Transfer USDT to the RWA-Provider f) Decrease the balancePerToken mapping The problem here lies within b), where the approval happens using the nominal USD value. If 1 RWA is worth 0.9 USD, it approves as example 90 RWA but the swapRwaForStbcMintEngineCp function attempts to transfer in 100 RWA: 1. Approval: IERC20(token).approve(address(_daoCollateral), amount) 2. TransferFrom: IERC20(address(rwaToken)).safeTransferFrom(msg.sender, address(_BUCKETS), amount * SCALAR_ONE_SIX / _ORACLE_USUAL.getPrice(rwaToken)); The approval is therefore insufficient, resulting in a function revert.
Recommendations	Consider converting and approving the correct nominal RWA amount which is being transferred in. Moreover we *highly* recommend to execute a one-time approval upon deployment, ensuring the DaoCollateral contract can always consume funds.

Comments / Resolution	

Issue	Incorrect condition check for LusDaoReward
Severity	Informational
Description	After a RWA offer has been executed, the minterReward is minted to the MinterPool and the amount of rewards is returned: returns (uint256 priceInUSD, uint256 lusdaoInstant, uint256 cpReward, uint256 minterReward) it is then fetched as follows: (, uint256 lusdaoReward,, uint256 cpReward)
Recommendations	First of all, we can observe a typo here, it should be minterReward, not cpReward (this does not impact the business logic). Now we realize the following condition-check: if (lusdaoReward > 0) Which is however wrong, lusdaoReward can be > 0 but minterReward can be still zero, rendering the rewardData creation useless.

Comments / Resolution	Consider use the correct variable naming and use the following condition check:
	
	if (minterReward > 0) {
	..
	}

Issue	Incorrect access control for swapRwaForStbcMintEngineMinter
Severity	Informational
Description	The swapRwaForStbcMintEngineMinter is solely invoked by the MintEngineExecutor upon the finalization of a provided RWA offer. However, the access control mechanism allows this function to be called by the MintEngineCpOrderbook contract as well: && !_REGISTRY_ACCESS.hasRole(MINT_ENGINE_CP_ORDERBOOK, msg.sender)
Recommendations	Consider removing this allowance.
Comments / Resolution	

MintEngineMinterPool.sol

Issue	_matchOrderbook uses actual USD value as parameter for _oracle.getQuote instead of nominal USDT amount
Severity	High
Description	The _matchOrderbook function is invoked upon the RWA offer creation, specifically by the matchMinterOffer function within the MintEngineMatching contract. Once we circle back to that execution, we realize that the amount parameter is the actual USD value for the RWA asset uint256 amount = rwaOracle.getQuote(tokenToExchange, amountToMatch) (uint256 amountMatched, bytes[] memory transferInfos) = mintEngineMinterPool.poolMatching(tokenPreferred, amount) —----------—----------—---------- Now if we further inspect the call-flow, we see how the amount variable is used: amount = _oracle.getQuote(token, amount); which is basically fetching the USD value for “amount” of USDT, while amount is the USD value of the RWA asset. **Let’s illustrate with an example why this is incorrect:** USDT price = 1.1e6 RWA Value (USD) = 100e6 The correct idea would be to fetch the USDT amount for the provided RWA value. However, the calculation within getQuote is as follows:

	price = (price * (amount)) / SCALAR_ONE_SIX -> 1.1e6 * 100e6 / 1e6 -> 110e6 In the codebase, this would mean 100 USD are equivalent to 110 USDT, assuming a price of 1.1$ per USDT, which is obviously incorrect. If USDT is worth 1.1 and we have 100 USD, this would mean the corresponding USDT amount is 100/1.1 = 90.9 This will not only attempt to take the incorrect USDT amount from the offer but also result in an incorrect condition check: || amountMatched == amount) as well as incorrect following arithmetic operations: uint256 remaining = amount - amountMatched
Recommendations	Consider converting the USD amount to the corresponding USDT amount.
Comments / Resolution	

Issue	Claim for unfinalized ID will result in loss of outstanding rewards
Severity	**High**
Description	Users can call the claim function to claim their rightful rewards for the specific ID. The problem is that this function can be invoked even if the virtual offer for the corresponding ID is not yet fully consumed. This will grant users only partial rewards depending on the leftover amount in the ID's offer, but will fully reset their balancePerOffer, effectively excluding them from the outstanding rewards for this ID. Furthermore, these funds will be permanently locked. In the previous audit, we included the following recommendation: "Consider updating the user balance after a claim. **It is important to ensure users can only claim once an epoch (currentId) has passed, otherwise users might lose tokens during the claim process.** A full re-audit of the whole mintEngine scope is highly recommended." Unfortunately, this recommendation was not followed.
Recommendations	Consider ensuring users can only claim once the virtual offer for the corresponding ID was completely consumed.
Comments / Resolution	

Issue	Incorrect amountMatched return value for _matchOrderbook call will create illegal offer in MintEngineCpOrderbook
Severity	**High**

Description	ǂ The _matchOrderbook function is invoked upon the matching mechanism for a RWA offer creation. More specifically, it handles the matching of existing minter offers with an incoming RWA offer.
	
	To understand the issue, we need to understand the two (normally three) scenarios for this function:
	
	a) The current virtual offer in the MinterPool can completely cover the incoming RWA offer
	
	b) The current virtual offer in the MinterPool cannot cover the incoming RWA offer
	
	The first scenario is straightforward, the incoming amount is matched with an offer, the offer is decreased and amountMatched is returned.
	
	The latter scenario is slightly more complex, the incoming amount is matched with an offer but this offer cannot completely cover the RWA offer, therefore it is deleted and a new offer with the leftover USDT amount in the contract created.
	This new offer is then matched with the remaining amount, consider the case where this offer can cover the remaining amount. The amountMatched variable is returned and then used in the following comparison check to create an offer in the MintEngineCpOrderBook with the leftover amount:
	
	if (amountMatched != amount) {
	mintEngineCpOrderbook.createOffer(
	caller, tokenToExchange, amount - amountMatched,
	tokenPreferred);
	}

	This logic so far is perfectly fine, the problem is just that the returned amountMatched does not incorporate the matched amount from the first offer match. Instead it is simply newly cached (overridden):
	
	uint256 remaining = amount - amountMatched;
	
	uint256 balanceToken = getBalancePerToken(token) - amountMatched;
	
	_createOffer(token, balanceToken);
	
	(offer, amountMatched) = _offer[token].take(remaining);
	
	This will result in creating a leftover offer due to an incorrect condition check based on the incorrect amountMatched variable while the amount can in fact be completely matched:
	
	if (amountMatched != amount) {
	
	mintEngineCpOrderbook.createOffer(caller, tokenToExchange, amount - amountMatched, tokenPreferred;
	
	}
	
	this offer is moreover not covered by any RWA tokens (they are completely matched):

Recommendations	Consider aggregating the amountMatched return value in the already existing amountMatched value instead of overriding it: (offer, newAmountMatched) = _offer[token].take(remaining); return (amountMatched += newAmountMatched, transferInfos)
Comments / Resolution	

Issue	Insufficient distinction for LusDAO allocation allows users to claim rewards from other users
Severity	High
Description	The claim function allows users to claim STBC and LusDAO. These rewards are beforehand added via the addReward function. (address token, uint256 amountUsUSD, uint96 offerId) = abi.decode(data[i], (address, uint96, uint96)); _token[usUSD].claimablePerOffer[token][offerId] += amountUsUSD; For the LusDAO portion, there is no distinction made between tokenUsed. For instance, it doesn't matter if the LusDAO reward is for USDT or USDC, the amount for the offerId will be increased for both. Users can therefore claim excess LusDAO, which will then result in a revert due to insufficient contract balance.
Recommendations	Consider implementing a distinction to which token the LusDAO is accounted for.
Comments / Resolution	

Issue	Violation of checks-effects-interactions pattern
Severity	Low
Description	Throughout the contract there are one or multiple spots which violate the checks-effects-interactions pattern, to ensure a protection against invalid states, all external calls should strictly be implemented after any checks and effects (state variable changes). The most notable spot is the claim function, specifically the balancePerOffer value is only reset post-transfer. If there are ever tokens with hooks used, this would result in a full-blown reentrancy vulnerability.
Recommendations	Consider following the CEI pattern.
Comments / Resolution	

Issue	Access control for deposit function is not sufficiently used
Severity	Informational
Description	The deposit function incorporates the onlyMintEngineOrderOrMatching modifier, which allows it to be called by MINT_ENGINE_ORDER and MINT_ENGINE_MATCHING. However, the deposit function is solely invoked by the MintEngineOrder contract and not by the MintEngineMatching contract.
Recommendations	Consider invoking the deposit function by the MintEngineMatching contract as well (for the case that the orderbook was insufficiently filled to cover the USDT offer.

Comments / Resolution	

Issue	Redundant condition checks within _matchOrderbook
Severity	Informational

Description	The _matchOrderbook function is responsible for matching an incoming RWA offer with funds in the minter pool. It does that by creating or taking virtual offers. Virtual offers cannot be canceled as they are naturally created by the contract itself.
	
	This means the following condition check is redundant:
	
	if (statusBefore != Queue.OfferStatus.WAITING) {
	// * If the order was not waiting we should not subtract the amount from the balance
	amountMatched = 0;
	}
	
	Virtual offers will either be waiting or simply overridden (_begin index is increased)

Recommendations	Consider removing these redundant condition checks.
Comments / Resolution	

Oracles

AbstractOracle.sol

Line by line check is missing for this contract.

A breaking change has been introduced: The price is not returned with 18 decimals but rather with the token's own decimals.

Issue	Price update in same block can be abused to harm the protocol
Severity	Informational
Description	The getPrice function is used on multiple occasions, specifically when depositing a RWA asset and redeeming STBC. The getPrice function is designed in such a way to allow different prices per block (it does still update if already fetched before in the block). That opens up the theoretical possibility where users can harm the protocol: a) Deposit 100 RWA, receive 100 STBC (price = 1) b) Price is updated to 0.9 c) Redeem 100 STBC, receive 111.1 RWA Users which are able to forge blocks, might be able to reproduce such a scenario. However, this issue is only rated as informational because such a practice is rather rare and only updating once per block could be even worse.
Recommendations	The specific solution for this problem would be to not update the price more than one time per block. However, we do not

	recommend that because ... .. if only updated once per block, a malicious user can inspect a price update (directly on the oracle), deposit in the same block (where the price was already fetched) and then execute another transaction in the consecutive block (where the new price is fetched). Example: 1. Price in block X is fetched with 1.0 2. Price is updated to 0.9 3. Deposit 100 RWA, receive 100 STBC (the price 1.0 is used) 4. Wait until next block 5. Redeem 100 STBC, receive 111.11 RWA Therefore, we do not recommend any change to the codebase but rather **keeping awareness of this**.
Comments / Resolution	

Issue	_oraclePriceChangeAboveMax may backfire
Severity	Informational
Description	The oracle incorporates the _oraclePriceChangeAboveMax, in the scenario where the current oracle price is more than 50% increased compared to the previous price, the oracle will be considered as untrusted and the lastGoodPrice is returned. In itself, this is a good mechanism to secure against faulty oracle price. However, just recently we witnessed the limitations of such a safeguard for CompoundV2: https://x.com/CharlesWangP/status/1761741654946451830?s=20 Therefore, such a design choice must be carefully thought-out. A positive fact is that the RWA in our scenario is a stablecoin, therefore, the likelihood of valid +50% price increases is very low.
Recommendations	Consider if this could happen for this protocol in practice, if yes, consider adjusting the price threshold.
Comments / Resolution	

Issue	_checkDepegPrice does not work with exotic tokens
Severity	Informational
Description	The _checkDepegPrice function only works for tokens with 18 decimals and 6 decimals. Tokens with different decimals will be inherently treated as 6 decimal tokens. This issue is only rated as informational since the protocol only uses 6 and 18 decimal tokens.
Recommendations	Consider applying a universal approach: ($.maxDepegThreshold * (10**decimals) / BASIS_POINT_BASE)
Comments / Resolution	

Issue	Protocol is effectively DoS'ed whenever a depeg happens
Severity	Informational
Description	During the getPrice function, if the asset is a stable asset, a depeg check is executed. In the scenario where a depeg happened, this will revert the price fetching mechanism, effectively DoS the protocol.
Recommendations	This mechanism is likely exactly as desired, however, this still exposes a risk which we need to include as an informational issue. No action other than acknowledging is necessary here.
Comments / Resolution	

Issue	targetDigit variable is unused
Severity	Informational
Description	The aforementioned variable was previously used, however, with the new getPrice function it is not used anymore.
Recommendations	Consider removing it.
Comments / Resolution	

ClassicalOracle.sol

Line by line check is missing for this contract.

DataPublisher.sol

Line by line check is missing for this contract.

UsualOracle.sol

Line by line check is missing for this contract.

Registry

RegistryAccess.sol

Line by line check is missing for this contract.

This contract has been heavily refactored, inheriting the AccessControlDefaultAdminRules contract and essentially just overriding the AccessControlDefaultAdminRules , allowing the DEFAULT_ADMIN to grant any role to any address as well as removing it.

No new issues found.

RegistryContract

Line by line check is missing for this contract.

No new issues found.

Token

ERC20Blacklist

Issue	Blacklist functionality prevents burnFrom
Severity	Low
Description	The ERC20Blacklist contract is used by the STBC and UsDao tokens. Both tokens expose the burnFrom function which allows privileged addresses to burn tokens from any address. The _beforeTokenTransfer function however will revert if tokens from an blacklisted address are burned. There is no possibility to burn tokens from such an address.
Recommendations	Consider if this will become an issue or is designed by intention. If undesired, consider fixing it but not allowing blacklisted users themselves to invoke the “burn” function.
Comments / Resolution	

ERC20Whitelist

Issue	UBO reset will not automatically remove whitelisted privilege
Severity	Low
Description	Within the RegistryContract, it is possible to set the UBO to zero. Only users with an UBO != 0 can be whitelisted. A problem can arise if a user then has the UBO removed again but the whitelisted privilege was not automatically removed.
Recommendations	Consider if this can expose a problem, if yes, consider additionally checking for the UBO correctness within the isWhitelisted function.
Comments / Resolution	

LsausUSD

The LsausUSD token was refactored and is using a very specific reward algorithm which is derived from the standard masterchef logic. The important difference is that this does not incorporate an accumulator variable, therefore, rewards are only partially distributed. We will illustrate this with an example:

Example 1: Standard MC algorithm

1. Alice has staked 100 tokens and is the only staker the rewardPerSecond variable is 1 token.
2. After 100 seconds, Alice has received 100 tokens.

Example 2: LsausUSD reward algorithm

1. Alice has staked 100 tokens, the maxSupply is 1000 token but Alice is the only staker, the rewardPerSecond variable is 1 token.
2. After 100 seconds, Alice has received 10 tokens.

As one can see, in the first example the whole reward allocation is fully consumed while in the second example the allocation is only consumed on a pro-rata share based on the maxSupply.

Issue	Lack of allowance check for transferFrom allows for stealing tokens from other users
Severity	**High**
Description	The transferFrom function lacks an allowance check. Anyone can just transfer tokens from any recipient to himself, resulting in a total loss of tokens.
Recommendations	Consider implementing OpenZeppelin's allowance check: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol#L156
Comments / Resolution	

Issue	Incorrect usage of bondStart bricks reward distribution
Severity	**High**

Description	Upon contract deployment, the bondStart time is determined. Usually, this is the time from when rewards are being distributed. However, rewards are not distributed from that time but rather immediately after contract deployment. This can be observed in the _getPendingReward function:
	
	uint256 blocksBetweenBondStartAndLastReward = block.number - lastRewardBlock;
	
	after the first deposit, lastRewardBlock is essentially set to the first deposit time:
	
	if ($.userInfo[account].lastRewardBlock == 0) {

	$.userInfo[account].lastRewardBlock = block.number; return 0;
	
	}
	
	Therefore, no matter what time the bondStart is set, it will not have any impact.
	
	Well, that in itself does of course not make this eligible as high risk issue, however, the following does:
	
	function totalBondBlocks() public view returns (uint256) {
	
	LsausUSDStorageV0 storage $ = _lsausdStorageV0();
	return ($.bondEnd - $.bondStart);
	
	}
	
	uint256 maxUsDao = $.rewardPerBlock * totalBondBlocks();
	
	The bondStart variable is essentially used for the calculation of how much is refunded after the close. If bondStart is now after the first deposit, this will result in a flawed calculation.

Recommendations	Consider incorporating the bondStart variable in the _updateReward function, simply not distributing any rewards as long as the bondStart time is not reached. This would then also need to be adjusted in the LsausUSDFactory to distribute rewards only for the bond block period.
Comments / Resolution	

Issue	Lack of bondEnd check within claimYield/_getPendingRewards distributes rewards even after bond has ended
Severity	High
Description	The claimYield function allows users to claim their yield even if the bond has ended. This mechanism in itself is correct, since otherwise users would lose rewards if not claimed on time. The problem is within the _getPendingRewards function, as this does not check for the bondEnd time to limit the reward allocation. If a user claims weeks after the bondEnd, he will still receive rewards for that whole time: uint256 blocksBetweenBondStartAndLastReward = block.number - lastRewardBlock uint256 rewards = (blocksBetweenBondStartAndLastReward) * balance * rewardPerBlock / maxSupply This will essentially steal rewards from users that have not yet claimed.
Recommendations	Consider implementing a check which limits the blocksBetweenBondStartAndLastReward calculation to bondEnd, if block.number is larger.
Comments / Resolution	

Issue	Leftover transfer to BucketDistribtion contract will permanently lock tokens
Severity	**High**
Description	The close function allows for withdrawing the leftover rewards to the BucketDistribution contract. However, there is no corresponding call which actually adds them to a bucket: function addAssetToBucket(bytes32 bucket, address token, uint256 amount) this will result in these tokens being locked permanently in the BucketDistribution contract.
Recommendations	Consider implementing a corresponding call to the addAssetToBucket selector.
Comments / Resolution	

Issue	Close will result in permanently stuck rewards due to incorrect calculation
Severity	**High**
Description	The close function calculates the leftover rewards as follows: uint256 leftOverUsDao = (maxUsDao * ($._maxSupply - $._totalMinted) / $._maxSupply) - SCALAR_HUNDRED_SZABO; If we apply the following example values: rewardPerSecond = 1e18 duration = 1000

maxUsDao = 1000e18
maxSupply = 1000e18
totalMinted = 500e18

Following the calculation, the leftover amount would be:

1000e18 *((1000e18 - 500e18) / 1000e18)

a very simple pro-rata calculation based on the minted amount on the max supply and the total rewards, resulting in
500e18 leftover tokens.

The problem with this calculation is that not from the first second on the “totalMinted” variable applies. This means, if for example there was only a mint amount of 100e18 for most of the time and 1 second before the end someone deposited 400e18 tokens, the leftover amount would be actually

1000e18 * ((1000e18 - 100e18) / 1000e18) = 900e18

(ignore the 1 second distribution for 500e18 tokens for simplicity reasons)

Therefore, the applied calculation will simply not work.

Moreover, the SCALAR_HUNDRED_SZABO seems to be pointless, as the calculation rounds already down.

Additionally, the following condition check:

if ($._maxSupply == $._totalMinted)

must also be removed, as this does not guarantee that all tokens have been consumed.

Recommendations	A correct fix would be to implement a tracker in how many rewards are actually distributed/allocated, however, that would require a big adjustment in the contract. Therefore, it might make most sense to implement a manual withdrawal function for the leftover rewards, which grants the owner the privilege over handling the rewards. While this is a centralization privilege, we do not see an issue here, since the full protocol heavily relies on governance.
Comments / Resolution	

Issue	Rewards will be permanently stuck if first deposit happens after bondStart
Severity	Medium

Description	Consider the above mentioned issue is fixed and reward distribution starts onwards bondStart.
	
	Another more or less critical problem is that unallocated rewards (bondStart is block = 100 and first stake is block =200) will not be allocated to users nor rescuable. This is due to the initial adjustment of lastRewardBlock on the first deposit:
	
	if ($.userInfo[account].lastRewardBlock == 0) {
	
	$.userInfo[account].lastRewardBlock = block.number; return 0;
	
	}
	
	*This issue with relation to bondStart assumes the distribution is fixed

	in the first place.
Recommendations	Consider either implementing a special accounting mechanism for these unallocated rewards which allows for withdrawing exactly these unallocated tokens or simply implement a rescue function to withdraw any potential unallocated rewards. The second solution seems more feasible in this environment given the potential complexity for the implementation of the first solution.
Comments / Resolution	

Issue	BUCKET_DISTRIBUTION can never claim rewarded LusDAO
Severity	Medium
Description	During a transfer, the transferabilityFee is transferred to the BUCKET_DISTRIBUTION contract. This means that this contract will be a token holder for a specific time, which makes it eligible for LusDAO rewards, which is also indicated in the update: _updateReward($, address(_BUCKET_DISTRIBUTION)); The problem is that this contract does not expose the claimYield function, which means that these allocated rewards can never be claimed, they are effectively lost.
Recommendations	A fix for this issue would be to implement a claimYield function to the BUCKET_DISTRIBUTION contract which allows to claim the allocated yield. However, this implementation would further alter the code, opening the doors for further vulnerabilities, since the further logic in the BucketDistribution would need to be implemented to actually use these funds.

	Therefore, we recommend simplifying the whole process by ignoring these allocated rewards and claim them manually with a recovery function after the bond has ended.
Comments / Resolution	

Issue	Incorrect order of operations within _transfer results in falsified pendingReward calculation
Severity	Medium
Description	Whenever a transfer is being executed, a transferabilityFee is taken and sent to the BUCKET_DISTRIBUTION address, the rewards for this address are however only updated after the transfer, which means that the received amount will be included in the hindsight reward amount, which then falsifies the whole reward allocation. This issue is only rated as medium vulnerability since there is no claimYield function within the said contract.
Recommendations	Consider simplifying the whole process by ignoring these allocated rewards and claim them manually with a recovery function after the bond has ended. The update should furthermore happen before the transfer (balanceOf increase).
Comments / Resolution	

Issue	Inconsistency in feeBucket validation
Severity	Low

Description	The feeBucket variable determines to which bucket the LsausUSD fee is added, during the setFeeBucket function, the following validations are existent:
	
	if (_BUCKET_DISTRIBUTION.isBucketCollateralized(bucket)) {
	revert InvalidBucket();
	}
	
	if (_BUCKET_DISTRIBUTION.getBucketToken(bucket) == address(this)) {
	revert InvalidToken();
	}
	
	These validations are non-existent in the constructor.

Recommendations	Consider keeping consistent and applying these validations within the constructor as well.
Comments / Resolution	

Issue	Usage of block.number instead of block.timestamp is discouraged
Severity	Low
Description	A few years ago, most masterchefs were using block.number instead of block.timestamp for reward based calculations. It quickly turned out that, based on the chain, there are minor differences in the block times. Over a larger time-frame, this could mean that one day more and one day less rewards are distributed. Taking this as a reason, developers started to modify their masterchefs to incorporate block.timestamp instead of block.time. This contract is using block.time instead of block.timestamp, which eventually slightly distorts rewards. For instance, if the blockchain stops producing blocks, users will not get any rewards during that time. The most recent time where this example has happened is exactly today on 23rd february, where AVAX C-Chain stopped producing blocks for a few hours.
Recommendations	In our opinion, a fix for this is not mandatory, however, it is still mandatory to mention this issue.
Comments / Resolution	

LusDAO

No new issues found.

Stbc

No new issues found.

usDAO

No new issues found.

Utils

Queue

Line by line check is missing for this contract.

Issue	SafeCast is redundant
Severity	Informational
Description	The usage of SafeCast for incrementional values, especially if starting from zero, is redundant. There will never be such a case where the _begin is larger than a uint128: uint128 idx = SafeCast.toUint128(uint256(queue._begin) + index);
Recommendations	Consider removing this redundant code.
Comments / Resolution	

Core

DaoCollateral

Line by line check is missing for this contract.

It is highly recommend for Spearbit to conduct fuzzing on the cross-contract interactions between the DaoCollateral and BucketDistribution contract on the addAssetToBucket and withdrawRWA functions to ensure the arithmetic operations work as intended and rounding does not expose any harm. Due to time-constraints this will not be part of our review.

Issue	swapForStbcMintEngineCp mints incorrect STBC amount
Severity	High
Description	The swapForStbcMintEngineCp function is invoked with amount = USD value of RWA amount by the MintEngineExecutor: *Emergence of amount parameter:* transferInfos = _appendToBytesArray(transferInfos, abi.encode(offer.token, matched * price / SCALAR_ONE_SIX, offer.owner) (address token, uint128 amount, address owner) = abi.decode(transferInfos[i], (address, uint128, address)); _daoCollateral.swapRwaForStbcMintEngineCp(owner, caller, token, amount); —————————————————— This is then correctly converted to the RWA amount which needs to be transferred in: IERC20(address(rwaToken)).safeTransferFrom(msg.sender,

	address(_BUCKETS), amount * SCALAR_ONE_SIX / _ORACLE_USUAL.getPrice(rwaToken));
	
	as well as added to the bucket:
	
	_BUCKETS.addAssetToBucket(BUCKET_RWA_DAO_COLLATERAL, rwaToken, amount * SCALAR_ONE_SIX /_ORACLE_USUAL.getPrice(rwaToken));
	
	The problem is that this calculation is also applied for the amount of STBC to be minted:
	
	_USUSD.mint(stableOwner, (amount * SCALAR_ONE_SIX / _ORACLE_USUAL.getPrice(rwaToken)) * 1e12);
	
	Which means that the USDT provider will not receive the corresponding USD worth but rather the nominal RWA amount, inflated by 1e12. While in fact the **USD value** of the RWA amount should be minted.

Recommendations	Consider minting the corresponding USD amount to the stable provider.
Comments / Resolution	

Issue	Price conversion for swapRwaForStbcMintEngineMinter is flawed
Severity	**High**
Description	Currently, the swapRwaForStbcMintEngineMinter function receives the matched USDT amount as amount parameter: *Call-path to demonstrate the emergence of the amount parameter:* (Queue.Offer memory offer, uint256 amountMatched) = _offer[token].take(amount); amountMatched != 0 ? transferInfos = _appendToBytesArray(transferInfos, abi.encode(token, uint96(amountMatched), address(this), uint96(offerId))) (uint256 amountMatched, bytes[] memory transferInfos) = mintEngineMinterPool.poolMatching(tokenPreferred, amount); (address token, uint96 amount, address owner, uint96 offerId) = abi.decode(transferInfos[i], (address, uint96, address, uint96)) (, uint256 lusdaoReward,, uint256 cpReward) = _daoCollateral.swapRwaForStbcMintEngineMinter(caller, owner, rwaToken, amountStable, amount); ————————————————————————— From the call-path above, we realize that this is the nominal USDT value which has been matched. (To the corresponding USD value of the RWA offer) The problem is that the _getPriceAndToken function uses the RWA value: priceInUSD = _getPriceAndToken(rwaToken, amount); while the amount is still the USDT value. This will return an incorrect priceInUSD, resulting in incorrect minted amounts of rewards.

	The reason why this issue is high instead of medium is that a user can potentially abuse this incorrect conversion and combine it with the issue: “Combination of dynamic price nature from lusDAO token and bypass of _redeemBalance increase can be used to profit from repetitive MintEngine execution and STBC redeems” in such a manner that even if the reward fee configured is significantly lower than redemption fee, a potential arbitrage profit can be created due to the flawed conversion, this would allow a malicious user to gain an unlawful amount of LusDAO.
Recommendations	An isolated fix for this issue would be to use the USDT token as parameter, ensuring the proper conversion.
Comments / Resolution	

Issue	Price conversion for swapRwaForStbcMintEngineCp is flawed
Severity	**High**
Description	Upon MintEngine flow, the swapRwaForStbcMintEngineCp function is invoked with amount = USD value of RWA: *Call-path to demonstrate the emergence of the amount parameter:* transferInfos = _appendToBytesArray(transferInfos, abi.encode(offer.token, matched * price / SCALAR_ONE_SIX, offer.owner)) (uint256 amountMatched, bytes[] memory transferInfos) = mintEngineCpOrderbook.orderMatching(tokenToExchange, amount) mintEngineExecutor.executeOrderCp(transferInfos, caller,

tokenToExchange)

(address token, uint128 amount, address owner) = abi.decode(transferInfos[i], (address, uint128, address))

_daoCollateral.swapRwaForStbcMintEngineCp(owner, caller, token, amount)

—----------—----------—----------

The problem lies in the _getPriceAndToken call within the swapRwaForStbcMintEngineCp function:

priceInUSD = _getPriceAndToken(rwaToken, amount)

this function is essentially calculating the USD value for “amount” of “rwaToken”. The problem here is that the provided amount parameter is already the USD value.

Therefore, this calculation will be flawed and an incorrect amount of usDAO/LusDAO rewards is minted.

This issue is only valid for the MintEngine path. For an offer cancellation within the MintEngineCpOrderbook, the amount value is the fee (nominal RWA amount).

The reason why this issue is high instead of medium is that a user can potentially abuse this incorrect conversion and combine it with the issue:

“Combination of dynamic price nature from lusDAO token and bypass of _redeemBalance increase can be used to profit from repetitive MintEngine execution and STBC redeems”

in such a manner that even if the reward fee configured is significantly lower than redemption fee, a potential arbitrage profit

	can be created due to the flawed conversion, this would allow a malicious user to gain an unlawful amount of LusDAO.
Recommendations	An isolated fix for this issue would be to simply remove this conversion, as the amount is already the USD value, this is perfectly fine. However, the cancellation call in the MintEngineOrderCp function must be adjusted as well.
Comments / Resolution	

Issue	swapRwaForStbcMintEngineMinter transfers USDT amount instead of RWA amount in
Severity	High
Description	The swapRwaForStbcMintEngineMinter transfers the RWA token to the bucket as follows: IERC20(address(rwaToken)).safeTransferFrom(msg.sender, address(_BUCKETS), amount) which will be also added to the bucket: _BUCKETS.addAssetToBucket(BUCKET_RWA_DAO_COLLATERAL, rwaToken, amount) However, the amount parameter is the matched USDT amount, which can be significantly different from the RWA amount. A corresponding issue has been created within the MintEngineExecutor: “executeOrderMinter uses nominal USDT amount for swap which

	results in incorrect amountMatched parameter for swapRwaForStbcMintEngineMinter call”
Recommendations	This issue must be fixed within the MintEngineExecutor contract, however, for consistency reasons it is mandatory to report this in the DaoCollateral contract as well.
Comments / Resolution	

Issue	RWA provider can unintentionally DoS the contract
Severity	Medium

Description	For both swap functions: swapRwaForStbcMintEngineMinter and swapRwaForStbcMintEngineCp, the following check is executed:
	
	if (_redeemBalance[ubo] > 0) {
	
	revert NotEnoughDeposit();
	
	}
	
	If a user has an existing RWA offer in the orderbook but then continues operating within the normal business logic, which can mean redeeming tokens, this will increase the _redeemBalance and will then result in a revert whenever the offer is matched.
	
	This can happen regularly with the constant need of governance intervention (closure of order).

Recommendations	This issue is part of the design choice by the usualUSD team. We highly recommend re-thinking the _redeemBalance design. We have already flagged multiple potential issues in the previous round which are related to this design.

Comments / Resolution	

Issue	_distributeUsDAO can revert due to underflow in arithmetic operation
Severity	Medium
Description	The _distributeUsDAO function is responsible for minting LusDAO to the rwa and stable providers, as well as minting usDAO to the protocol. It receives two uint256 as parameters: uint256 amount uint256 usdAmount Whereas amount is a pro-rata share from the usdAmount: amountToReward, amountToReward * usDAORatio / SCALAR_TEN_KWEI The usdAmount is then further used to calculate the share towards the RWA and stable providers, as well using a pro-rata calculation. Afterwards, the aggregated amount for the stable and rwa provider is deducted from the amount parameter: uint256 bucketAmount = amount - userReward If userReward is now larger than the amount, this will revert. Since this will only happen in a rare scenario where cpLusDAOReward and minterLusDAOReward are > 100%, this issue is marked as medium.

Recommendations	Consider explicitly returning early whenever userReward >= amount.
Comments / Resolution	

Issue	Redeem fee rounds in favor of the user
Severity	Low
Description	Whenever users redeem STBC, a part of the STBC gets distributed to the BucketDistribution. The fee calculation is as follows: stableFee = amount * redeemFee / SCALAR_TEN_KWEI This will round against the favor of a user, taking less fee than it should. In that scenario, depending on the reward calculation it is likely not a big issue, however, if the reward is exactly the same amount as the fee, one could trick this calculation to receive the full reward but pay a slightly lower fee. Several projects have been exploited due to the exactly same issue. A similar issue exists within the swapRwaForStbcMintEngineCp function where the RWA amount to be transferred in is rounded down: IERC20(address(rwaToken)).safeTransferFrom(msg.sender, address(_BUCKETS), amount * SCALAR_ONE_SIX / _ORACLE_USUAL.getPrice(rwaToken)) This could open up another potential exploit window, however, due to time-constraints of this review we are unable to create a PoC for that scenario.

Recommendations	Consider rounding against the user, in that example that would mean the fee calculation must round up. The RWA amount must also be rounded up.
Comments / Resolution	

Issue	**<_getTokenAmountForAmountInUSD will not work for tokens with 6 decimals**
Severity	Informational

Description	The _getTokenAmountForAmountInUSD function is responsible for calculating the RWA amount based on the burned STBC amount. The STBC amount is here "hard-coded" as 1 USD, this is a design choice by the protocol.
	
	The problem arises here:
	
	returnedCollateral = _getTokenAmountForAmountInUSD(burnedStable, collateralToken);
	
	amountInToken = _getQuoteInToken(amount, token);
	
	the parameters are:
	
	a) burned STBC amount (post-fee)
	b) RWA
	
	(uint256 priceInUSD, uint8 decimals) = _getPriceAndDecimals(token)
	
	This will now return the USD price for the RWA (in 18 decimals) and the RWA decimals (6).

	return amount * 10 ** decimals / priceInUSD If now the amount is denominated with 6 decimals, this would result in the following outcome: 100e6 * 1e6 / 1e18 = 0 This issue is only rated as informational since STBC is 18 decimals in V0.
Recommendations	For V0, we do not see it necessary to adjust the codebase, however, if any time in future in any other version the stablecoin will have != 18 decimals, this calculation should be adjusted.
Comments / Resolution	

TokenMapping

Line by line check is missing for this contract.