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	Meuna
Website	meuna.io
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/meuna-io/meuna-contract/tree/3387119362ff8468617ef05e592fb2de2d92a8b3/contracts
Resolution 1	https://github.com/meuna-io/meuna-contract/tree/e8ed1ac543843410916eb52cf8df5f55096f9555/contracts

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	6	5		1
Medium	11	5		6
Low	9	3		6
Informational	10	2		8
Governance	7			7
Total	43	15		28

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

Global

Issue_01	Governance Privilege: Proxy upgradeability
Severity	**Governance**
Description	The whole architecture is meant to be deployed as a proxy contract, enabling the proxy admin to upgrade all logic contracts at will. This can result in a total loss of funds.
Recommendations	Consider incorporating a Gnosis Multisignature contract as proxy owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged, the client will incorporate a multisig.

Issue_02	Architecture does not work with transfer-tax tokens
Severity	**Informational**
Description	This protocol is not compatible with transfer-tax tokens. If these token types are used for any purpose within the contract, this will result in down-stream issues and inherently break the accounting.
Recommendations	Consider not using these tokens.
Comments / Resolution	Acknowledged.

Issue_03	Lack of SafeERC20 usage
Severity	**Informational**
Description	The contract uses the standard ERC20 pattern for transfers/approvals. This will malfunction for tokens that do not follow the IERC20 interface, for example those that return false or do not return a boolean on the transfer/approve call.
Recommendations	Consider either only using standard ERC20 tokens or implementing SafeERC20
Comments / Resolution	Acknowledged.

Config

AssetConfig

The AssetConfig contract is a simple registry contract which keeps track of all assets and their corresponding parameters. It is used within the Mint contract to determine the collateralization ratio and the liquidation discount.

The contract owner can add any asset to the system with its corresponding minCollateralizationRatio and auctionPremium.

The denomination of these settings is as follows:

a) minCollateralizationRatio = 1e18 (120e18 = 120% minimum collateralization)
b) auctionPremium = 1e18 (10e18 = 10% premium)

These settings are useful to incorporate the risk factor of an asset. Risky assets with higher volatility should have a higher collateralization ratio.

Privileged Functions

- transferOwnership
- renounceOwnership
- setConfigs

Issue_04	Governance Privilege: Incorrect setting of parameters can harm the protocol
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. This includes the setting of important parameters such as the decrease of the minCollateralRatio or the increase of the auctionDiscount. Incorrect setting of these could result in undesired liquidations, inefficient liquidations and bad debt in the form of unbacked assets.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged, the client will incorporate a multisig.

CollateralConfig

The CollateralConfig contract is, similar to the AssetConfig contract, a simple registry contract which keeps track of all collateral tokens and their corresponding parameters. The main setting for a collateral is the so-called multiplier, which is denominated in 1e5.

The multiplier is used together with the minCollateralRatio of an asset to determine the minimum collateralization. It is useful to adjust for more/less risky assets, a risky collateral asset can simply be marked with a higher multiplier, which indicates a higher minimum collateralization.

Privileged Functions

- transferOwnership
- renounceOwnership
- setConfigs

Issue_05	Governance Privilege: Incorrect setting of parameters can harm the protocol
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. A sudden increase of a multiplier can result in undesired liquidations.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged, the client will incorporate a multisig.

Oracle

PriceOracle

The PriceOracle is a simple registry contract that keeps track of tokens and their corresponding prices. The contract owner can set any arbitrary price to any arbitrary token.

Privileged Functions

- transferOwnership
- renounceOwnership
- setPrices

Issue_06	Governance Issue: Arbitrary price setting
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. The contract owner can simply set an arbitrary price and harm the protocol.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged, the client will implement an off-chain mechanism for price updates.

Issue_07	Oracle usage is impractical
Severity	**High**
Description	The current setup of the oracle is just a call to the setPrices function which then hardcodes the latest provided price. This practice is insufficient if it is not updated each block. Moreover, the oracle is prone to frontrunning.
Recommendations	Consider re-thinking the current oracle setup
Comments / Resolution	Acknowledged, the client will implement an off-chain mechanism for price updates which updates the interval every x seconds.

PriceOracleSyncPyth

The PriceOracleSyncPyth contract is a handler contract for the Pyth oracle. The contract owner can add and configure price feeds, there are two potential settings:

a) Returning the price in USD scaled with 18 decimals: ETH/USD

b) Returning the reversed price scaled with 18 decimals USD/ETH

Within the architecture, usually the first option will be chosen.

Additionally, to the price fetching mechanism via a price feed, the owner can simply arbitrarily set the prices via the setPrices function, this should be rather used as emergency function than during the normal business logic.

Privileged Functions

- transferOwnership
- renounceOwnership
- addPrice
- update
- setPrices

Issue_08	Governance Issue: Arbitrary price setting
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. The contract owner can simply set an arbitrary price and harm the protocol.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged, the client will implement an off-chain mechanism for price updates.

Issue_09	Oracle usage is impractical and prone to frontrunning
Severity	**Medium**
Description	The current oracle setup expects the update function to be called regularly in an effort for the project to work flawlessly. However, the developer indicated that this will only happen every hour, which is simply insufficient . Additionally, this execution is prone to frontrunning
Recommendations	The front-running issue can not be easily avoided and is inherently part of all oracles. A fix would be to refactor this logic and implement a grace period which incorporates a deposit initiation and deposit execution with the final price. Additionally to this, the oracle must be updated in the smallest possible interval (the smaller the better).
Comments / Resolution	Acknowledged, the client will implement an off-chain mechanism for price updates which updates the interval every x seconds.

Issue_10	Setup will not work with return values that have more than 18 decimals
Severity	**Medium**
Description	Whenever the oracle returns a value with > 18 decimals, this will simply not work as it will unsafely cast the negative value to uint32, resulting in an overflow. For instance, if the return value is denominated with 19 decimals, this means that we need to scale the price down by 1 decimals, however, _expo will be 4294967295, instead of -1.
Recommendations	Consider either rewriting the architecture to properly handle this scenario or ensure that never an oracle with more than 18 decimals is used.
Comments / Resolution	Resolved, the protocol now enforces this requirement.

Issue_11	Reverse approach will not work for very low priced tokens
Severity	**Informational**
Description	The standard approach is to return the asset value in USD, however, the reverse approach is to return the USD value in asset. This is done by dividing 1e36 with the return value (scaled by 18 decimals). This approach will not work if the price is very low, resulting in the division product becoming zero.
Recommendations	Due to the fact that such a low-priced asset is very very rare and likely not used throughout the protocol, we do only consider this issue as informational and do not recommend any change.
Comments / Resolution	Acknowledged.

Token

MeunaAsset

The MeunaAsset is a simple ERC20 implementation which is used throughout the protocol as the mintable asset.

It incorporates standard logic such as burn and mint which can be executed by so-called setters. These setters can be determined by the contract owner, however in practice there should be only one setter which is the MintSynUpgrade contract.

Privileged Functions

- transferOwnership
- renounceOwnership
- addSetter
- removeSetter
- mint
- burn

No issues found.

Core

MintSynUpgrade

The MintSynUpgrade contract forms the main contract for the core protocol. While the main functionality of this contract is similar to that of a CDP (minting an asset by providing collateral), there are several different functionalities. Below we will explore all possibilities.

First of all, it needs to be mentioned that the configuration is completely abstracted to the AssetConfig and CollateralConfig contracts. These two contracts will essentially handle which assets can be minted and which tokens can serve as collateral as well defining the minCollateralRatio and auctionDiscount for an asset and the multiplier for a collateral token.

The contract therefore exposes the following functionalities:

1. Open a normal position: This allows users to provide collateral and mint an asset. Users can freely determine how much assets to receive for the provided collateral value, as long as the position is deemed as sufficiently collateralized.

2. Open a short position: This allows users to provide collateral, which then mints the asset to the Short contract. This asset is then immediately swapped and locked within the Locker contract. Additionally, it will create a staking position on behalf of the position owner in the StakingContractUpgrade contract.

3. Deposit collateral to a position: This allows users to add collateral towards a position and will simply increase the collateral balance and the position collateralization.

4. Withdraw collateral from a position: This allows users to withdraw collateral from a position, ensuring that the position is still sufficiently collateralized after the withdrawal. If a position is fully withdrawn, which is the case when there is no outstanding debt and the full collateral is removed, the position is marked as closed and will be removed. Additionally if it is a short position that is fully withdrawn, it will clear out any existing lock.

5. Mint assets from a position: This allows users to mint an asset from a position, ensuring the position is still sufficiently overcollateralized once the asset has been minted. For a short position

it simply follows the same pattern as in the position opening: It mints the asset to the Short contract, swaps it and locks it in the Locker contract, furthermore it will increase the position in the Stake contract for the size of the minted asset.

6. Repay minted asset: This allows users to repay/burn an asset for a position to increase the collateralization. For a short position it will additionally decrease the position in the Staking contract.
Whenever assets are burned, this will decrease the collateral of the position by a fee based on the burned asset amount.

7. Auction: This allows users to liquidate an undercollateralized auction and receive the collateral with a premium. Debt will be paid back and the collateral will be taken from a user's position. If a short position is liquidated, it will automatically decrease a user's stake by the repaid amount.

The locked amount for a short position can be withdrawn once the unlock time has been surpassed, this will prevent any further increase of the short position.

***This contract is only meant to be used with 18 decimal tokens.**

Privileged Functions

- transferOwnership
- renounceOwnership
- setAssetConfig
- setCollateralConfig
- setCollector
- setFee
- setShortContract
- pause
- unpause

Issue_12	Governance Privilege
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. The contract owner can for example change the assetConfig and collateralConfig contracts.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_13	Lack of closure check within auction call allows to disrupt user storage
Severity	**High**
Description	The auction function allows for partially or fully liquidating an undercollateralized position, this call will always invoke the removePosition function whenever the position is fully liquidated (ie. leftCollateralAmount = 0 or leftAssetAmount = 0). The problem with removePosition is that if it is called multiple times with the same positionId, it will corrupt the storage for the user. Due to the fact that there is currently no check if the position is already closed, it is possible to call the auction function on a positionId that has already been withdrawn or was already liquidated. As mentioned already, this will completely disrupt the storage for a user.
Recommendations	Consider reverting if the position is already marked as closed.

Comments / Resolution	Resolved.

Issue_14	Combination of several issues and occurrences allows users the creation of almost risk-free and permanent staking position
Severity	**High**
Description	The difference between this issue and between “Bad-debt situation can result in users to unlawfully keep their stake position” lies within the fact that this can be actively exploited while the other issue can just coincidentally happen. Whenever a position is liquidated, this means that the liquidator will receive the collateral for a discount. As already explained, this is handled by increasing the provided asset value. The auctionDiscount is determined by the minimum of auctionDiscount for the asset OR the minCollateralRatio. This means if the auctionDiscount is 25% but the minCollateralRatio is 20%, it will use the minCollateralRatio. The idea behind this is to not automatically create bad debt whenever the position is undercollateralized. However, the usage of the exact minCollateralRatio also does not seem too practical as this means liquidations will almost always result in bad-debt if positions are not immediately liquidated once they have reached their minimum collateralization (in fact, a liquidation is already possible once the exact threshold is reached - not crossed). This behavior will therefore always result in some staking positions not being withdrawn, granting users permanent rewards. This can even be abused by users for their own benefit (depending on the fee).

Due to the fact that the oracle is pull-based, a malicious user can frontrun the oracle update and profit from it:

Illustrated:

asset.auctionDiscount = 25e18
asset.minCollateraRatio = 120e18
collateral.multiplier = 1e5
asset value = 1e18
collateral value = 1e18

1. Alice deposits 120e18 collateral tokens

... oracle update happens, increasing asset price by 10% .. Alice frontruns this oracle update ...

2. Alice takes out 100e18 - 1 wei of assets as loan and creates a short position with it
-> Alice receives a stake position of 100e18 -1
-> Alice's position is at this exact moment not liquidatable

3. The oracle update has happened Alice's position is now liquidatable and Alice backruns the update and liquidates her own position with another wallet

-> asset value = 1.1e18
-> collateral value = 1e18
-> position.mintAmount = 100e18-1
-> position.collateralAmount = 120e18

-> assetPriceInCollateral = 1.1e18
-> assetValueInCollateral = (100e18-1) * 1.1e18 / 1e18
= 110e18 (rounded up for simplicity reasons)

	-> Alice repays 100e18-1 assets with her second wallet -> auctionDiscount = 120e18 / 100 -1e18 = 2e17 -> discountPrice = 1.1e18 * 1e18 / (1e18-2e17) = 1.375e18 -> assetValueInCollateralDiscount = (100e18-1) * 1.375e18 / 1e18 = 137.5e18 -> refundAssetAmount = (137.5e18 - 120e18) * 1e18 / 1.375e18 -> 12.72e18 -> Alice's second wallet receives 12.72e18 assets back -> Alice's second wallet receives 120e18 of collateral (minus fees applied) -> The stake position is decreased by 87.28 Alice effectively received her full collateral (minus fee) back and was able to create a permanent stake position of 12.72e18 tokens which grants her yield.
Recommendations	This issue is a prime-example of what happens if several other issues are combined and exploited maliciously: 1. Bad-debt does not reduce stake position 2. Oracle prone to frontrunning 3. Usage of minCollateralRatio as cap for auctionDiscount Ideally, all of these issues are fixed. However, as already mentioned the fix of issue 1 requires refactoring of the code. Therefore, we are of the opinion that it is a good practice to reduce the auctionDiscount in the scenario of a minCollateralRatio usage by a few percent. Or simply consider just setting a reasonable auctionDiscount which will never be close to the minCollateralRatio. Together with a reasonable fee, this seems to make this attack economically uninteresting. Of course, at the end of the day it depends on the price volatility.

	This issue can be completely considered as fixed if the recommendation for “Bad-debt situation can result in users to unlawfully keep their stake position” is implemented.
Comments / Resolution	Resolved, if there is no collateral left, the whole short stake within the staking contract is being withdrawn.

Issue_15	Bad-debt situation can result in users to unlawfully keep their stake position
Severity	**Medium**
Description	As already explained within the “Auction can result in unhealthier position post-liquidation” issue, it is possible for a liquidation to not fully pay back the minted asset. This has the background that it is always intended to pay out the full discount and in such a scenario as explained in this issue, the repayment amount will be reduced to match the exact underlying collateral. This will result in an insufficient decrease of the staking position for the position owner. This issue is intensified due to the fact that for this scenario, it is possible for users to liquidate themselves (under rare circumstances), which means there won’t be any nominal loss for a user besides the fee which is applied on the collateral.
Recommendations	Consider fully decreasing the stake position in the scenario where the liquidation results in leftCollateralAmount == 0.
Comments / Resolution	Resolved, if there is no collateral left, the whole short stake within the staking contract is being withdrawn.

Issue_16	Liquidation can result in bad debt for the protocol
Severity	**Medium**
Description	LIquidations are designed in such a way to adjust the provided asset value in order to reflect the discount. This means if a user provides an asset with a value of 1e18 and the discount is 20%, the asset value will be adjusted to 1.25e18. For comparison, other lending protocols such as AAVE just repay the debt without any price adjustments and then apply the discount on top of the collateral repayment: https://github.com/aave/aave-v3-core/blob/master/contracts/protocol/libraries/logic/LiquidationLogic.sol#L498 Furthermore, the full discount is paid to a user no matter if the actual collateral can cover this. This is achieved in such a manner to eventually decrease the repayment amount if the collateral cannot cover it. Therefore, it is possible for a position to be fully out of collateral but the mintAmount to be non-zero. This will happen if positions are not immediately liquidated once they are considered undercollateralized, if they further decrease their collateralization ratio, the auctionDiscount can become larger than the overcollateralized security deposit, thus resulting in insufficient assets being repaid but the full collateral taken. **Illustrated** collateral = 2 USD asset = 1 USD -> position.collateralAmount = 100e18 -> assetAmount = 190e18 -> asset.auctionDiscount = 15e18

-> assetValueInCollateral = 0.5e18

minimum collateralization = 120%
actual collateralization = 105%

Since this position is heavily undercollateralized, it can get liquidated.

a) Alice calls auction with 190e18

b) Undercollateralization check passes:
->assetPriceInCollateral * minimumCollateralization * multiplier / 1e25
< pos.collateralAmount
-> 95e18 * 1e5 * 120e18 / 1e25 = 114e18
-> actual collateralAmount = 100e18

Therefore, this check passes and the position can get liquidated

c) Calculate auction discount:
-> asset.auctionDiscount/100
-> 15e18 / 100 = 1.5e17

d) Calculate discount price:
-> assetPriceInCollateral * 1e18 / (1e18-discount)
-> 0.5e18 * 1e18 / (1e18 - 2e17) = 0.625e18

e) Calculate repaymentValue adjusted by discount:
-> burnAmount * discountPrice
-> 190e18 * 0.625e18 / 1e18 = 118.75e18

f) Since 118.75e18 is more than the existing collateral of 100e18, we must calculate the excess provided asset amount and refund it:
-> refundAssetAmount = repaymentValue - collateralAmount * 1e18 / discountPrice
-> (118.75e18 - 100e18) * 1e18 / 0.625e18 = 30e18

	g) This means Alice has provided 160e18 of asset and will receive 100e18 of collateral. The protocol experiences a bad debt of 30e18 assets
Recommendations	This is a design choice made by the developers. Inspired by other lending protocols, a solution would be to reduce the premium in such a scenario to not exceed the outstanding collateral. However, also this design choice may have side-effects as it may result in less incentives for liquidators to liquidate such positions. At the end of the day, it depends on the Meuna team to decide on which design to use. Such a refactoring would require an additional check besides the resolution round which will result in a nominal fee, as Bailsec's resolution round only covers fixes which don't require refactoring or the addition of new logic.
Comments / Resolution	Acknowledged.

Issue_17	Auction function incorrectly allows liquidations for positions which are on collateralization threshold
Severity	**Medium**

Description	Whenever a new position is opened, collateral is withdrawn or assets are minted, the collateralization check is executed in such a way to allow these executions if the position is exactly meeting the minimum collateralization threshold or above:
	
	require(assetConfig.getMinCollateralRatio(asset).mul(collateralConfig.getMultipier(collateral)).div(1e5) <= collateralRatio,"low collateral ratio than minimum");
	
	if((assetValueInCollateral.mul(mulipier).mul(assetConfig.getMinCollateralRatio(pos.asset))).div(1e25) > collateralAmount){
	revert("Cannot mint asset over than min collateral ratio");
	}
	
	if((assetValueInCollateral.mul(mulipier).mul(assetConfig.getMinCollateralRatio(pos.asset))).div(1e25) > collateralAmountAfterSub){
	revert("Cannot withdraw collateral over than minimum collateral ratio");
	}
	
	This is perfectly fine as in fact a position should not be considered as unhealthy as long as the minimum collateralization is met.
	
	Within the auction function, this check is however not consistent:
	
	if(assetValueInCollateral.mul(collateralConfig.getMultipier(pos.collateral)).mul(assetConfig.getMinCollateralRatio(pos.asset)).div(1e25) < pos.collateralAmount){ revert("Cannot liquidate a safely collateralized position");
	}
	
	As one can see, auction apparently allows for the liquidation of a

	position which exactly meets the collateralization threshold. This is incorrect and can lead to unlawful liquidations. **Illustrated:** asset.minCollateralRatio = 120e18 collateral.multiplier = 1e5 assetPrice = 1e18 collateralPrice = 1e18 1. Alice deposits 120e18 collateral tokens and provides a collateralRatio of 120e18: -> uint256 mintAmount = ((amount.mul(collateralPriceInAsset)).mul(100).div(collateralRatio)) -> 120e18 * 1e18 * 100 / 120e18 = 100e18 -> Alice receives 100e18 mint tokens 2. Bob recognizes Alice's position and invokes the auction function: -> safety check: -> 100e18 * 1e5 * 120e18 / 1e25 < 120e18 = false -> The position can be successfully liquidated Additionally, it is clear to mention that rounding/truncation plays a big role in all arithmetic operations. This means even if the check is changed to be consistent in all operations, the possibility exists that a position can still be liquidated after it has been created.
Recommendations	Consider being consistent and only allowing liquidations once the collateralization falls below the minimum collateralization. Additionally, consider thinking about a grace value implementation for openPosition / mintAsset / withdraw, which does not allow users to decrease the collateralization up to the exact threshold. This could for example be 1%.

Comments / Resolution	Resolved without a threshold implementation.

Issue_18	Usage of minCollateralRatio as auctionDiscount cap is flawed
Severity	**Medium**
Description	Whenever a liquidation happens, the auctionDiscount is determined as follows: Math.min(assetConfig.getAuction(pos.asset).div(100),assetConfig.getMinCollateralRatio(pos.asset).div(100).sub(one)); The idea behind this logic is to ensure that the discount is not larger than the minimum collateralization. Illustrated this means if we have a minCollateralRatio of 120e18, we can have 120e18 as collateral and 100e18 tokens as debt (as long as they are worth the same). Therefore, a full liquidation should not take more than 120e18 from the collateral. This is exactly the rationale behind this cap. The execution is however incorrect, as this value is used to increase the asset price rather than discount the collateral price, which leads to a larger discount than expected. **Illustrated:** a) Alice has a position with 120e18 collateral and 100e18 assets. Both are worth 1 USD and the auctionDiscount is 25e18 b) The position now becomes unhealthy and gets liquidated. Bob calls the auction function with 100e18+1 wei as burnAmount. The auctionDiscount is now fetched which is based on the minCollateralRatio because this is smaller than the auctionDiscount for

	this asset: 120e18 / 100 - 1e18 = 2e17 discountedPrice = 1e18 * 1e18 / (1e18-2e17) = 1.25e18 c) Bob therefore provides an amount of 100e18+1 assets but receives 125e18 of the collateral. This is simply incorrect as this would result in bad debt. The mathematical flaw lies within the fact that the minCollateralRatio is used to decrease the divisor to scale up the asset price, which then results in an incorrect discountPrice. This issue is specifically critical because it allows the *"Combination of several issues and occurrences allows users the creation of almost risk-free and permanent staking position"* issue, to be executed completely risk free.
Recommendations	**Solution 1:** Consider calculating the correct discountPrice, which would be roughly 1.6666666e17 for a minCollateralRatio of 20e18. However, this would mean the logic needs to be refactored to actually incorporate fetching the correct minimum auctionDiscount between auctionDiscount and collateralRatio. Because if auctionDiscount is 19e18 and collateralRatio is 120e18, it would fetch 1e19, which still results in bad debt in our example. **Solution 2:** Optionally, this issue can be disregarded if it is ensured that the auctionDiscount is always set to a reasonable low value which ensures that the usage of auctionDiscount will never result in bad-debt whenever a position has just fallen below the minimum collateralization.

	Ie. use an auctionDiscount of 5e18 for an asset with a minCollateralRatio of 120e18.
Comments / Resolution	Acknowledged, the client decides to move on with solution 2, using the following values: - 20% discount - 150% min collateralization

Issue_19	Liquidations will revert if position owner is blacklisted from collateral asset
Severity	**Low**
Description	Whenever a full liquidation is happening, any eventual collateral which is not transferred to the liquidator (collateral - assetValueInCollateralDiscount) is being transferred to the position owner. Additionally, if there is a lock in the Lock contract, this lock is released and also transferred to the position owner. In such a scenario where the owner is blacklisted for the collateral asset, transfers and thus the auction call will revert.
Recommendations	Consider simply not using these assets as collateral.
Comments / Resolution	Acknowledged.

Issue_20	Dust debt repayment will prevent full liquidations
Severity	**Low**
Description	Whenever the auction function is called with burnAmount = position.mintAmount, it will liquidate the full position (if allowed). Position owners can frontrun this call, repaying 1 wei which then results in a revert.
Recommendations	Consider downsizing the burnAmount to position.assetAmount if it is larger than the minted asset amount for this position.
Comments / Resolution	Acknowledged.

Issue_21	Lack of validation for several setter functions
Severity	**Low**
Description	The contract has several setter functions, namely: a) setAssetConfig b) setCollateralConfig c) setCollector d) setFee e) setShortContract None of these setter functions has validation, which allows for setting the zero address or an unreasonably high fee. In the worst scenario, this could result in repaying assets making the position more unhealthy because of a large fee.
Recommendations	Consider validating these functions appropriately.

Comments / Resolution	Resolved.

Issue_22	Lack of decimal normalization
Severity	**Low**
Description	Throughout the contract there are arithmetic operations that involve two or more assets. In such a scenario where one asset will have 6 decimals and another asset will have 18 decimals, these operations will result in a flawed outcome. This issue is rated as low because the protocol is intended to work only with 18 decimals tokens. If under any circumstance other tokens are added, this will not work anymore and this issue is considered as **high risk**.
Recommendations	Consider either acknowledging this issue and never tokens with != 18 decimals as collateral and assets or add additional logic to normalize tokens always to 18 decimals.
Comments / Resolution	Resolved, the protocol enforces that only tokens with 18 decimals can be used.

Issue_23	Incorrect rounding throughout the protocol
Severity	**Informational**
Description	Throughout the protocol there are several instances where the asset amount is translated to the collateral value or vice-versa. Notably this is done on the following interactions: a) MintSynUpgrade.openPosition (translate collateral to asset) b) MintSynUpgrade.withdraw (translate asset to collateral) c) MintSynUpgrade.mintAsset (translate asset to collateral) d) burnAsset (translate asset to collateral) e) MintSynUpgrade.auction (translate asset to collateral) Most instances simply round in the incorrect direction, for example, within the withdraw function, the borrowed (asset) amount is translated to the collateral value. This operation rounds down, thus slightly displaying lower debt than the user actually has. The correct operation would be to convert the collateral to the corresponding asset value, as this would then operate against the favor of the user. A simple demonstration would be for the withdraw function: **Illustrated:** position.mintAsset = 10 assetPrice = 0.1e18 collateralPrice = 2e18 Therefore, the calculation: uint256 assetValueInCollateral =

	assetAmount.mul(assetPrice).div(colleteralPrice); -> 10 * 0.1e18 / 2e18 = 0.5 results in assetValueInCollateral of zero, effectively allowing to withdraw all collateral while leaving 10 wei of bad debt.
Recommendations	At Bailsec, we often weigh the impact of an issue against a possible issue introduction during the fix. Whenever we are of the opinion that an issue has no real negative impact on the business logic, we recommend the client to rather acknowledge it instead of fixing it, because with further code changes, the risk of implementing more severe issues is not worth the fix of “cosmetical” issues. This can be applied to this issue as well, simply because there is no exploitable fact behind having displayed a few wei less debt than the user actually has.
Comments / Resolution	Acknowledged.

Issue_24	Configurational setting of multiplier and minCollateralRatio is vulnerable to incorrect setting
Severity	**Informational**
Description	Whenever a position is created, collateral is withdrawn, asset is minted or a liquidation attempt is made, it will consult the minCollateralRatio of the asset and the multiplier of the collateral. This configuration is important to adjust the platform risk based on the volatility of the asset and as well can be used to incentivize users providing collateral, by increasing the multiplier. We need to understand two important points: a) These values are interconnected, which means together they form the needed collateralRatio. If the multiplier is decrease, this will automatically increase the capital efficiency / decrease the possible collateralRatio b) There is no limitation in which asset can be minted with which collateral. This essentially means you can mint Asset A with collateral A as well as with collateral B If we now combine these two points, we will quickly realize that this configurational environment exposes risks. Consider a scenario where you add Asset A with minCollateralRatio of 120e18 and add Collateral A with multiplier = 1e5. This allows minting Asset A with a minimum collateralization ratio of 120%. This configuration seems fine. However, now we add Asset B with minCollateralRatio of 200e18 and Collateral B with multiplier = 7e4. The configuration in itself for this pair is also perfectly fine, as it determines the minimum collateralization ratio with 140%.

	The problem is now b), as it is possible to mint Asset A with Collateral B, which has the following settings: - minCollateralRatio = 120e18 - multiplier = 7e4 results in a minimum collateralization ratio of 84%, thus making the protocol drainable. This illustration showcases the importance of a correct governance surveillance, as this would not be the first time that such an issue occurs. Additionally, the environment is vulnerable to this issue.
Recommendations	Consider being careful when setting these parameters.
Comments / Resolution	Acknowledged.

Issue_25	Contract is only meant to be used with External-Owned-Addresses
Severity	**Informational**
Description	Currently, most functions implement an onlyEOA modifier which only allows external owned addresses to interact with the protocol. This means smart contract wallets can never interact with the protocol.
Recommendations	Given that this is a likely a security driven decision, we do not have any further recommendations.
Comments / Resolution	Acknowledged.

ShortContractUpgrade

The ShortContractUpgrade is an intermediate contract which is invoked by the Mint contract whenever a short position is created, adjusted or liquidated. It incorporates logic to swap the minted asset to collateral, locks the collateral in the Locker and creates a stake position on the Staking contract.

Privileged Functions

- transferOwnership
- renounceOwnership
- setAssetPool
- setMintContract

Issue_26	Governance Privilege: PID can be changed
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, the PID of an asset or the mintContract could be changed which then disrupts the business logic.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged. a multisig will be used.

Issue_27	Swap can be brutally sandwich-attacked
Severity	**High**
Description	Currently, there is no minAmountOut parameter specific for the swap. This can result in a large loss for short position openings/increases if the execution is sandwich-attacked.
Recommendations	Consider adding a minAmountOut parameter to the openPosition and deposit functions which then is passed to the openShort and increaseShort functions (in case of a short position, otherwise it can be disregarded) and then applied on the swap.
Comments / Resolution	Resolved.

Issue_28	afterAuction function is unused
Severity	**Informational**
Description	The aforementioned function is never used. Unused code can confuse third-party reviewers.
Recommendations	Consider removing the unused function.
Comments / Resolution	Resolved.

LockUpgrade

The LockUpgrade contract is a simple locker contract which locks the temporary collateral from a short position. We will quickly re-iterate the short position creation:

Alice opens a short position on the MintSynUpgrade contract. Therefore she provides collateral and receives an asset. This asset will then be sold back to the collateral and the received collateral is locked in this contract. Afterwards a stake in the StakingContractUpgrade contract is created from the size of the minted asset.

The Locker contract therefore simply takes custody of the received collateral until either the position is repaid with external funds and fully withdrawn, liquidated or the unlock time has been reached.

Whenever assets for a short position are minted, this will also increase the lock position in the exact same manner as during the position opening.

Privileged Functions

- transferOwnership
- renounceOwnership
- setDuration
- setShortContract
- pause
- unpause

Issue_29	Locker is only compatible with one token
Severity	**High**
Description	The Locker contract is responsible for storing the collateral which was received during the creation or increase of a short position. Upon position creation/increase, the openShort/increaseShort functions on the Short contract are called: https://github.com/meuna-io/meuna-contract/blob/3387119362ff8468617ef05e592fb2de2d92a8b3/contracts/ShortContractUpgrade.sol#L40 https://github.com/meuna-io/meuna-contract/blob/3387119362ff8468617ef05e592fb2de2d92a8b3/contracts/ShortContractUpgrade.sol#L48 As one can see, these functions will swap the asset to the corresponding collateral with the Locker as receiver contract. It is clear that this protocol will support multiple different tokens as collateral. Notably, there is only one deployed Locker contract in the architecture and always the same contract will receive different collateral tokens. (This can be simply verified by following the flow) The problem: The Locker is only compatible with one collateral: https://github.com/meuna-io/meuna-contract/blob/3387119362ff8468617ef05e592fb2de2d92a8b3/contracts/LockUpgrade.sol#L16 Now there are two possibilities: a) A malicious user frontruns such a short position opening (of an

	incompatible token) with sending a few wei of the correct collateral token to the Locker. This will result in the transaction succeeding and the users collateral will be permanently lost in the Locker. b) The transaction will just revert, making it impossible to create short positions with any other token than this main collateral token.
Recommendations	The Locker needs to be refactored and re-audited. It should be compatible with multiple collateral tokens. Additional testing is necessary to prevent such issues in the future.
Comments / Resolution	Resolved, it will always swap to the same collateral token. This will alter the initial business logic.

Issue_30	Manual unlock will result in corrupted user storage
Severity	**High**
Description	The storage for user positions is exactly handled as within the MintSynUpgrade contract. This means that duplicate calls to the removePosition function will alter a user's storage in the exact same way. In itself, it wouldn't be a problem if there is no such scenario to invoke this function more than 1 time. However, due to the fact that the unlockPosition function invokes the removePosition function without simultaneously remove the position from the MintSynUpgrade contract, there are several instances where this flow can result in a problem: a) LockUpgrade.unlockPosition + MintSynUpgrade.withdraw b) LockUpgrade.unlockPosition + MintSynUpgrade.auction c) LockUpgrade.unlockPosition + LockUpgrade.unlockPosition ***Provided is just one of many examples of what could go wrong:*** —---------- it.only("ATTACK TEST", async function() { await stable.mint(user1.address, "1000000000000000000000") await oracle.setPrices([asset00.address, asset01.address, stable.address], ["1000000000000000000", "5000000000000000000", "1000000000000000000",]) await collateralConfig.setConfigs(

```
            [stable.address, asset01.address],
            [
                ["stableMock", "100000", true],
                ["asset01", "100000", true],
            ]
        )
        // min ratio 130%
        await assetConfig.setConfigs(
            [asset00.address],
            [["mAPPLE", "20000000000000000000",
"13000000000000000000", true]]
        )
        // min ratio 150%
        await assetConfig.setConfigs(
            [asset01.address],
            [["mTESLA", "20000000000000000000",
"15000000000000000000", true]]
        )
        let mintSynTest = await upgrades.deployProxy(MintSynUpgrade, [
            oracle.address,
            assetConfig.address,
            collateralConfig.address,
            "100",
            collector2.address,
        ])
        await asset00.addSetter(mintSynTest.address)

        //open position 1
        await stable
            .connect(user1)
            .approve(mintSynTest.address,
"100000000000000000000")
        await mintSynTest
            .connect(user1)
            .openPosition(
```

```
            "1000000000000000000",
            asset00.address,
            stable.address,
            "1500000000000000000",
            false
        )
    let pos = await mintSynTest.positions(1)
console.log("--Minted position 1--")
    console.log(
        "user 1  active positions:",
        await mintSynTest.countPosition(user1.address)
    )

    //open position 2
    await asset01
        .connect(user1)
        .approve(mintSynTest.address,
"1000000000000000000")
    await mintSynTest
        .connect(user1)
        .openPosition(
            "1000000000000000000",
            asset00.address,
            asset01.address,
            "1500000000000000000",
            false
        )
    pos = await mintSynTest.positions(2)
    mintAmountPos2 = pos.mintAmount
    console.log("--Minted position 2--")
    console.log(
        "user 1  active positions:",
        await mintSynTest.countPosition(user1.address)
    )
```

```
        // Pay off position 2
        await asset00
            .connect(user1)
            .approve(mintSynTest.address, mintAmountPos2)

        await mintSynTest.connect(user1).closePosition(1)
        console.log("--Closed position 2--")
        console.log(
            "user 1  active positions:",
            await mintSynTest.countPosition(user1.address)
        )

        // Liquidate position 2
        await oracle.setPrices(
[asset00.address, asset01.address],
            ["115000000000000000000",
"500000000000000000000"]
        )

        await asset00
            .connect(user2)
            .approve(mintSynTest.address, "700000000000000000000")

        await oracle.setPrices(
            [asset00.address, asset01.address],
            ["116000000000000000000",
"500000000000000000000"]
        )

        await mintSynTest.connect(user2).auction(1, "0")

        // Check test
        console.log(
            "--Liquidated the already closed position 2 with 0 amount--"
        )
```

	console.log("user 1 active positions:", await mintSynTest.countPosition(user1.address)) }) 1. Position 1 is created, with 150% CR 2. Position 2 is created 150% CR 3. Position 2 is closed 4. Position 2 is liquidated 5. Contract shows the user has no more positions, forgetting about position 1. Thus position 1 cannot be liquidated or withdrawn from. Step 4 can be carried out by anyone. So malicious users can affect normal users who close their positions. **Critical Severity** —---------- If `removePosition()` is called twice on the same position Id, the first invocation will set the `mapIndex` of that position. mapIndex[positionId] = 0 Here 0 is actually a legitimate index. So the second invocation will end up deleting the 0th element of the `userPositions[user]` array, which could be a completely unrelated lock. This can lead to lost funds. After further investigation, we came to the conclusion that the removePosition call in the unlockPosition function is redundant and does not serve any purpose.
Recommendations	Consider removing the removePosition call within the unlockPosition function. It is sufficient if a position is removed whenever it is fully closed or liquidated.
Comments / Resolution	Resolved.

Issue_31	unlockPosition function can be called multiple times
Severity	**Low**
Description	As already indicated above, the unlockPosition function can be called multiple times with the same positionId. If the “Manual unlock will result in corrupted user storage” function is fixed, there is no critical outcome from invoking this function two times with the same positionId. However, in our mission to make codebases as secure as possible, we always recommend limiting redundant user flexibility. IMPORTANT: This should not prevent the releasePosition function from being called.
Recommendations	Consider marking a position as “manuallyUnlocked” and revert if a position is marked as such already.
Comments / Resolution	Resolved.

Issue_32	Change of duration may inadvertently impact position owners negatively
Severity	**Low**
Description	Whenever a position is created, its unlockTime is set based on the block.timestamp and the current duration. If a position is increased, the unlockTime is reset in the same manner. A duration increase may negatively impact position owners if they increase their position.
Recommendations	This issue can be safely acknowledged if it is a design choice. If however this is unintended, consider caching the duration upon position creation and apply this cached value upon position increase.
Comments / Resolution	Acknowledged.

Issue_33	(Full) Liquidation will revert if lock owner is blacklisted for the collateral token
Severity	**Low**
Description	During the full liquidation of a short position, the locked collateral is released (transferred out). If a position owner is blacklisted for the collateral, this transfer will revert, preventing liquidations
Recommendations	Consider not using tokens with a blacklist feature.
Comments / Resolution	Acknowledged.

Issue_34	Zero-transfers may revert for specific tokens
Severity	**Low**
Description	The releasePosition function (once the flow is fixed), will eventually attempt to do a zero-value transfer if the position was already unlocked. Zero-value transfers will revert for specific custom tokens, thus preventing a full withdrawal/liquidation.
Recommendations	Consider not using these tokens or wrapping the transfer into an if-clause which is only triggered once unlockAmount is non-zero.
Comments / Resolution	Acknowledged.

Issue_35	Receiver should not be changeable
Severity	**Informational**
Description	The increaseLock function will change the received based on the provided parameter. In the current implementation this is not an issue because the parameter will always be the same owner. However, if that is not the case under any circumstances and the receiver is changed, this will corrupt the storage later once the removePosition function is invoked with the new receiver as parameter. The reason behind this issue is that the new receiver in fact does not have a storage slot for this position.
Recommendations	Consider not changing the receiver upon the increasePosition call.
Comments / Resolution	Acknowledged.

Issue_36	Pause functionality remains unused
Severity	**Informational**
Description	None of the functions in the contract are using the whenNotPaused modifier, therefore the pausable inheritance is redundant. It is also not needed to implement it because the pausable safeguard is already incorporated in the Minter contract.
Recommendations	Consider removing the PausableUpgradeable inheritance.
Comments / Resolution	Resolved.

StakingContractUpgrade

The StakingContractUpgrade contract is a customized staking contract which is heavily inspired by Sushiswap's Masterchef contract:

https://github.com/sushiswap/masterchef/blob/master/contracts/MasterChef.sol

It allows users to stake tokens in different pools and distributes rewards among all stakers. The contract owner can add new pools, adjust the reward rate arbitrarily and modify pool allocations.

Additionally to the standard Masterchef logic, the contract exposes special pools for short positions, the only possibility to get an allocation in these pools is via opening a short position through the Mint contract or mint assets from a short position.

Privileged Functions

- transferOwnership
- renounceOwnership
- add
- set
- setMeunaPerSecond
- addSetter
- removeSetter
- setShortContract
- pause
- unpause
- updateWeight (setter)
- updatePerSec (setter)
- updateWeightAndPerSec (setter)

Issue_37	Several functions will alter rate and allocation in hindsight
Severity	**Medium**
Description	The contract exposes several functions which change the reward rate, change a pool's allocation or change the overall allocation, namely: a) updateWeight b) updatePerSec c) updateWeightAndPerSec d) add e) set f) setMeunaPerSec While a-c update the pool state only for the provided pools in the input parameters, which is already insufficient, d-f do not update the pool state at all. This will completely alter reward allocation in hindsight from the latest updated time onwards.
Recommendations	Consider invoking massUpdatePool at the beginning of all these functions and marking it as public instead of external and use the poolInfo.length as in the example below: https://github.com/sushiswap/masterchef/blob/master/contracts/MasterChef.sol#L112
Comments / Resolution	Acknowledged.

Issue_38	Add function lacks a check for the upper limit
Severity	**Medium**
Description	As mentioned in the issue above, it is mandatory to update all pool states before the reward rate is changed, weights are altered or the total weight is increased. Therefore, it is necessary to implement an upper limit to the add function, ensuring that massUpdatePool will not run out of gas.
Recommendations	Consider setting a reasonable upper limit, such as a maximum of 50 pools.
Comments / Resolution	Acknowledged.

Issue_39	Reward payout will revert if contract is insufficiently seeded
Severity	**Medium**
Description	Currently, whenever a deposit/withdrawal is happening, rewards are paid out automatically. In such a scenario where the contract has insufficient funds, the whole deposit/withdrawal flow will revert, including a potential cross-contract call to withdraw from short positions. This issue is inflated due to the fact that the contract does not expose an emergencyWithdraw function.
Recommendations	A short term solution would be to manually ensure that the contract is always sufficiently funded. A long term solution would be to implement additional functionality that stores the outstanding balance in an additional mapping

	whenever the contract balance is insufficient to pay out rewards. A simple example can be found in Trader Joe's implementation: https://github.com/traderjoe-xyz/auto-pool-token-farm/blob/main/src/APTFarm.sol
Comments / Resolution	Acknowledged, the contract will be sufficiently funded.

Issue_40	Lack of emergencyWithdraw function
Severity	**Medium**
Description	The emergencyWithdraw function became an important standard in masterchef contracts. The rationale behind this function is to pay out the stake while ignoring any reward update / reward payout. This can be specifically useful if the reward update or payout is bricked, which then allows users to still withdraw their funds. Such a special scenario can occur if the totalAllocPoint is zero and therefore the updatePool function reverts due to a division by zero error.
Recommendations	Consider implementing an emergencyWithdraw functionality.
Comments / Resolution	Resolved.

Issue_41	Reward token as pool token may result in loss of funds
Severity	**Medium**
Description	Currently, there is no such check that prevents the rewardToken from being added as a pool token. In such a scenario it may inadvertently happen that user staked tokens are distributed as reward tokens if the contract has insufficient reward tokens.
Recommendations	Consider adding a check that prevents the rewardToken from being added as a pool token.
Comments / Resolution	Resolved.

Issue_42	Lack of input validation for PID
Severity	**Low**
Description	The updatePool but also all other functions lack input validation for the correctness of a PID, this allows certain functions to be called with a non-existent PID (such as updatePool).
Recommendations	Consider validating all external functions, ensuring that the PID is not out of the range.
Comments / Resolution	Acknowledged.

Issue_43	Redundant rewardDebt increase during increaseShort
Severity	**Informational**
Description	The first rewardDebt adjustment during the increaseShort function is redundant as it is correctly updated afterwards anyways.
Recommendations	Consider removing the first rewardDebt update.
Comments / Resolution	Acknowledged.