HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Midas
Date: 25 Sept, 2023

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for Midas
Approved By	Niccolò Pozzolini | Solidity SC Auditor at Hacken OÜ Carlo Parisi | Solidity SC Auditor at Hacken OÜ
Tags	ERC20 token;
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://midas.app
Changelog	06.09.2023 - Initial Review 25.09.2023 - Second Review

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table of contents

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Introduction

Hacken OÜ (Consultant) was contracted by Midas (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

stUSD is an ERC20 backed by US treasuries. It is not a rebase tokens, instead the accrued value is computed by aligning the token's value with the one of iShares Treasury Bond 0-1yr UCITS ETF during the deposit and redeem operations. Deposit and redeem operations are taken care of by the vault contracts DepositVault and RedemptionVault. Only greenlisted addresses can perform deposit and redeem operations.

Operations rely on oracles but involve human intervention, thus the protocol is highly centralized.

Privileged roles

- *GREENLIST_OPERATOR_ROLE*: can blacklist users to prevent them to send and receive stUSD tokens.
- *BLACKLIST_OPERATOR_ROLE*: can greenlist users to let them perform deposit and redeem operations.
- *ST_USD_MINT_OPERATOR_ROLE*: can mint stUSD tokens.
- *ST_USD_BURN_OPERATOR_ROLE*: can burn stUSD tokens.
- *ST_USD_PAUSE_OPERATOR_ROLE*: can pause the stUSD token contract to block token transfers.
- *DEPOSIT_VAULT_ADMIN_ROLE*: can perform administrative operations on the DepositVault contract, such as:
    - withdraw ERC20 tokens in the vault
    - set the user fees for deposit operations
    - add and remove payment tokens (USD stablecoins)
    - pause the contract to block users' deposit requests
- *REDEMPTION_VAULT_ADMIN_ROLE*: can perform administrative operations on the RedemptionVault contract, such as:
    - withdraw ERC20 tokens in the vault
    - set the user fees for redeem operations
    - add and remove payment tokens (USD stablecoins)
    - pause the contract to block users' redeem requests
- *GREENLISTED_ROLE*: user able to perform deposit and redeem operations
- *BLACKLISTED_ROLE*: user unable to perform stUSD token transfers

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are provided.
- Technical description is complete.

Code quality

The total Code Quality score is **10** out of **10**.

- The development environment is configured.

Test coverage

Code coverage of the project is **100%** (branch coverage)

- Deployment and basic user interactions are covered with tests

Security score

As a result of the audit, the code contains no issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **10**. The system users should acknowledge all the risks summed up in the risks section of the report.

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
6 September 2023	4	4	2	5
25 September 2023	0	0	0	0

Risks

- The documentation is provided, but it is often wrong and with a lot of mistakes and confusion between deposit/redemption stUSD/USD, there are also undocumented features, such as a mapping where metadata are stored for the stUSD token. These metadata do not present security

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

risks, but they are not documented and it is hard to estimate what they will be used for.

- The protocol is highly centralized; using real world assets always requires a certain amount of centralization. In this case, the degree of centralization is very high, and the administrators of the protocol are expected to perform actions that could be automated with oracles, such as calculating the amount to mint. In this case, human error could be dangerous for users.
- The contracts are upgradeable; this audit will be valid only if the current version of the contracts gets deployed.
- The administrators of the protocol might set a 100% fee to deposit or withdraw the tokens from the various vaults.
- Since the min redemption amount can be updated by the protocol admins, users with deposited funds might require an additional deposit to be able to redeem the tokens.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items considered:

Item	Description	Status	Related Issues
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed	
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed	
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.	Passed	
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed	
Unchecked Call Return Value	The return value of a message call should be checked.	Passed	
Access Control & Authorization	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed	
SELFDESTRUCT Instruction	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant	
Check-Effect-Interaction	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed	
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed	
Deprecated Solidity Functions	Deprecated built-in functions should never be used.	Passed	
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.	Not Relevant	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Passed	
Authorization through tx.origin	tx.origin should not be used for authorization.	Passed	
Block values as a proxy for time	Block numbers should not be used for time calculations.	Not Relevant	
Signature Unique Id	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Not Relevant	
Shadowing State Variable	State variables should not be shadowed.	Passed	
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant	
Incorrect Inheritance Order	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Passed	
Calls Only to Trusted Addresses	All external calls should be performed only to trusted addresses.	Passed	
Presence of Unused Variables	The code should not contain unused variables if this is not justified by design.	Passed	
EIP Standards Violation	EIP standards should not be violated.	Passed	
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed	
User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Passed	
Data Consistency	Smart contract data should be consistent all over the data flow.	Passed	

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Flashloan Attack	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used. Contracts shouldn't rely on values that can be changed in the same transaction.	Passed	
Token Supply Manipulation	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Passed	
Gas Limit and Loops	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed	
Style Guide Violation	Style guides and best practices should be followed.	Passed	
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Passed	
Environment Consistency	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed	
Secure Oracles Usage	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Passed	
Tests Coverage	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed	
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Passed	

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Findings

■■■■ Critical

C01. Undocumented Feature

Impact	High
Likelihood	High

Proper documentation is vital for ensuring that developers, auditors, and users understand how to interact with the contract correctly and avoid potential vulnerabilities or misuse.

The two functions manuallyRedeem() are not documented, and their behavior deviates from the protocol's intended flow. These two functions are also related to issue C02.

Currently, these functions behave like an admin function to burn tokens without interacting with the user's mapping *requests*. The first function looks less permissive since it interacts with the oracle to convert amountStUsdIn to amountUsdOut, but since amountStUsdIn is not validated in any way, the two functions pose the same security risk. Anyway, the fact that one of the functions is not relying on an oracle is uncalled for and suspicious - the lack of oracle usage is further discussed in issue C04.

Path: ./contracts/RedemptionVault.sol : manuallyRedeem(address user, address tokenOut, uint256 amountStUsdIn), manuallyRedeem(address user, address tokenOut, uint256 amountStUsdIn, uint256 amountUsdOut)

Recommendation: Add clear documentation about the mentioned functionality or remove the function.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

C02. Undocumented Feature

Impact	High
Likelihood	High

Proper documentation is vital for ensuring that developers, auditors, and users understand how to interact with the contract correctly and avoid potential vulnerabilities or misuse.

The internal function _manuallyRedeem() is already part of the undocumented flow described in issue C01, and it contains one more undocumented and suspicious piece of logic.

Instead of burning the stUSD tokens from the user, it burns the tokens from the RedemptionVault contract itself. It is hard to make

sense of such a feature, so the client is advised to provide extensive documentation about it.

Path: ./contracts/RedemptionVault.sol : _manuallyRedeem()

Recommendation: Add clear documentation about the mentioned functionality or remove the function.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

C03. Requirement Violation

Impact	**High**
Likelihood	**High**

While being part of an unknown logic flow (see issues C01 and C02), the NatSpec of the function _manuallyRedeem() clearly states that such a function should take care of transferring the tokens to the user.

The mentioned functionality is missing.

Path: ./contracts/RedemptionVault.sol : _manuallyRedeem()

Recommendation: Implement the missing functionality, adjust the NatSpec, or remove the manual functions.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

C04. Requirement Violation, Off-Chain Logic Should Be Moved On-Chain

Impact	**High**
Likelihood	**High**

The two vaults of the protocol take care of converting stablecoins to stUSD and stUSD to stablecoins. To perform such actions, an oracle is used, but not all the time.

The function RedemptionVault.fulfillRedemptionRequest() is ignoring the values of amountStUsdIn provided by the user in the redemption request. Instead, the conversion to USD is performed off-chain and the USD value is provided as a function parameter and not validated in any way.

The same issue arises in RedemptionVault.manuallyFulfill(), DepositVault.fulfillDepositRequest, DepositVault.manuallyDeposit().

Such behavior is particularly weird since the correct flow is properly adopted in the function _getOutputAmountWithFee() implemented in both the vaults DepositVault and RedemptionVault.

Path: ./contracts/RedemptionVault.sol : fulfillRedemptionRequest(), _fulfillRedemptionRequest()

./contracts/DepositVault.sol : fulfillDepositRequest(), _fulfillDepositRequest()

Recommendation: Move the price conversion on-chain using the already implemented function _getOutputAmountWithFee().

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

C05. Undocumented Feature

Impact	**High**
Likelihood	**High**

Proper documentation is vital for ensuring that developers, auditors, and users understand how to interact with the contract correctly and avoid potential vulnerabilities or misuse.

The two functions manuallyDeposit() are not documented, and their behavior deviates from the protocol's intended flow.

These two functions can be used by the admin to mint stUSD tokens with no limits and without interacting with the user's mapping *requests*, thus breaking the stUSD backing.

The first function looks less permissive since it interacts with the oracle to convert amountUsdIn to amountStUsdOut, but since amountUsdIn is not validated in any way, the two functions pose the same security risk. Anyway, the fact that one of the functions is not relying on an oracle is uncalled for and suspicious - the lack of oracle usage is further discussed in issue C04.

Path: ./contracts/DepositVault.sol : manuallyDeposit()

Recommendation: Add the missing documentation about the mentioned functions or remove them from the code.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

■■■ High

H01. Highly Permissive Role Access

Impact	**High**

Likelihood	Medium

The role ST_USD_BURN_OPERATOR_ROLE is able to burn any amount of stUSD from any user without any confirmation from the users.

The protocol is rightfully centralized in many steps to be able to perform operations, but in this case, it is not required and should be avoided.

Path: ./contracts/stUSD.sol : burn()

Recommendation: Adopt a signature validation to make sure the user is allowing the burn operation

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

H02. Undocumented Feature

Impact	Medium
Likelihood	High

The function _validateAmountUsdIn() checks that the deposited amount is greater than the minimum one set by the admin.

This check is skipped if the depositing user has already deposited some funds. This behavior is neither expected nor documented. A user could have already redeemed the first tranche before depositing again, thus the limit should be enforced.

Path: ./contracts/DepositVault.sol : _validateAmountUsdIn()

Recommendation: Add the missing documentation about the said functionality or remove it from the code.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

Medium

M01. Misleading Naming

Impact	Medium
Likelihood	Medium

The name stUSD to describe the protocol's token is misleading. While being appealing to refer to staked USD, this is not what this token is representing. stUSD token is tracking the value of the iShares Treasury Bond 0-1yr UCITS ETF, and its value is currently 106$. This

lack of clarity would impact both code readability and the token's DeFi applications.

The function _getOutputAmountWithFee() in both of the vaults could also have a more clear name since it represents the output amount minus the fees.

Path: ./contracts/stUSD.sol : burn()

Recommendation: Since the purpose of stUSD is also to be adopted on external DeFi protocols, its name should not misuse a standard ('st' prefix for staked).

The adoption of a clearer name for the function _getOutputAmountWithFee() would improve the code's readability.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

M02. Missing Validation, Uncapped Min Redeem Amount

Impact	**High**
Likelihood	**Low**

In the contract RedemptionVault, the variable minUsdAmountToRedeem is set without being validated or bound in any way.

This could lead to a malicious funds lock from the protocol owner.

Path: ./contracts/RedemptionVault.sol : setMinAmountToRedeem()

Recommendation: Implement a maximum value for minUsdAmountToRedeem. An appropriate value could be the minimum amount to deposit in DepositVault contract.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

M03. Initializers Are Not Disabled On the Implementation Contracts

Impact	**High**
Likelihood	**Low**

It is a bad practice to leave a contract uninitialized.

An uninitialized contract can be taken over by an attacker. This applies to both a proxy and its implementation contract, which may impact the proxy.

To prevent the implementation contract from being used, the function _disableInitializers() should be invoked in the constructor to automatically lock it when it is deployed.

Path: ./contracts/RedemptionVault.sol : setMinAmountToRedeem()

Recommendation: Disable the initialization of the implementation contracts by invoking _disableInitializers() in the constructor.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

M04. Use safeTransfer Instead of Transfer For ERC20

Impact	High
Likelihood	Low

In the contract ManageableVault, in the function withdrawToken(), the transfer() function is used to send an ERC20 token.

This is a bad practice because the transfer() function does not check for errors while sending ERC20 tokens; the safeTransfer() function from the SafeERC20 library of OpenZeppelin does, and it is a much safer alternative to catch errors.

Path: ./contracts/ManageableVault.sol : withdrawToken()

Recommendation: Use safeTransfer() instead of transfer() to interact with ERC20 tokens.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

Low

L01. Missing Fee Validation

Impact	Medium
Likelihood	Low

Fees are not validated to be <= 100%, in this case 10000.

Setting a fee higher than 100% would result in a buffer overflow in function _getOutputAmountWithFee(), which would block users' redemption request - and also deposit requests if it was not for issue C04.

It is a good practice to also bound the fees to a reasonable value below 100%, to enhance users safety.

Path: ./contracts/abstract/ManageableVault.sol : setFee()

Recommendation: Bound the fees to 100%. Capping the fees to a reasonable value below 100% would make the protocol even safer for the users.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

L02. Redundant Mathematical Operation

Impact	Low
Likelihood	Low

The constant PERCENTAGE_BPS is assigned 100, but every time it is used, it gets multiplied by an additional 100, increasing operations Gas cost for no benefit.

Path: ./contracts/abstract/ManageableVault.sol : PERCENTAGE_BPS

Recommendation: Assign 10000 to the constant PERCENTAGE_BPS.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

L03. Redundant Struct Field

Impact	Low
Likelihood	Low

The two vaults RedemptionVault and DepositVault implement a struct each to accommodate information about user requests.

Both of the structs DepositRequest and RedemptionRequest include the field bool exists. Such a field is redundant and causes Gas overheads any time that variable is written.

Instead of relying on that variable, another field like user could be used to perform the check.

Path: ./contracts/DepositVault.sol : DepositRequest

./contracts/RedemptionVault.sol : RedemptionRequest

Recommendation: Remove the exists struct field and perform the existence check on another struct field, which is always set, like user.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

L04. Floating Pragma

Impact	Low
Likelihood	Low

The project uses floating pragmas `^0.8.0`.

This may result in the contracts being deployed using the wrong pragma version, which is different from the one they were tested with. For example, they might be deployed using an outdated pragma version, which may include bugs that affect the system negatively.

Path: ./contracts/*

Recommendation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment. Consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

Informational

I01. Typo in The NatSpec

In the contract DepositVault.sol, there is the initialize() function with comments that should explain the function and the parameters used in the function. In the comments, there is a typo: “_minAmountToDepositInEuro init. value for minUsdAmountToRedeem” should be instead “_minAmountToDepositInEuro init. value for minAmountToDepositInEuro”.

Typos can create confusion about the functionalities of a contract.

Path: ./contracts/DepositVault.sol : initialize()

Recommendation: Fix the typo to aid the readability of the contract.

Found in: 5f99226

Status: Fixed (Revised commit: 6079eb1)

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	**Critical**	**High**	**Medium**
Medium Likelihood	**High**	**Medium**	**Low**
Low Likelihood	**Medium**	**Low**	**Low**

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/RedDuck-Software/midas-contracts
Commit	5f9922683e2eb91d04cf2c9d610722241c8a0a8d
Requirements	Midas doc
Contracts	File: contracts/DepositVault.sol SHA3: 4c2ac72bfd1cc07a30bf5a7aad50495fd50edf46dd6467b5dd32feb2ec9600d3 File: contracts/RedemptionVault.sol SHA3: 17c22ad04e5c70b15066c0cf24a64ba27ed37c80fbc9d7ec59fc29e68959f5ea File: contracts/stUSD.sol SHA3: f5d74b11a9876721c1b442a4e467f102b0f1d55215fb9dc9dc5e43a11d07f7ce File: contracts/abstract/ManageableVault.sol SHA3: 20f969f58c0797049c0e2f0a52abef9604fa7c74899327976cf96ad164c8d4dc File: contracts/access/Blacklistable.sol SHA3: 74c10f75f2208c7aa53bdf60b4e38ced375e14a0888e4a73e3f297642d012af2 File: contracts/access/Greenlistable.sol SHA3: c35afbf1c1daffec15c10128bb54ef64ad65c122fe3e64c0808b4ba3cedf01e0 File: contracts/access/MidasAccessControl.sol SHA3: 6942211c30e97fcb54c2271c6cefcd37d01544d0cb1b3cc236b6026cf148bc89 File: contracts/access/MidasAccessControlRoles.sol SHA3: 37c5b95da82cc9456720a49050b1cde4183f342371b4709e136862b0b35933c7 File: contracts/access/Pausable.sol SHA3: 883ad172649f54d18070fddd6a7ab74c49075e7af037542abce09e6266c79d03 File: contracts/access/WithMidasAccessControl.sol SHA3: 9dc0ed76ae9cd2777a954f1dc10e94d2f43a4169429249f3dcd8f17a4701a2a9 File: contracts/feeds/DataFeed.sol SHA3: 1727802dba54a5a482268c600d2e9e44545ad6704852042847e065f9363ebc71 File: contracts/interfaces/IDataFeed.sol SHA3: 961bb8991c85a68d86eb1811e81f8baaa1863bb2f7f1ea54ebd23d292d2eea5a File: contracts/interfaces/IDepositVault.sol SHA3: a5e3957985ed705e10f48b3b47f73495379c7053f3d8b2c9d7b7a2eb5239da7a File: contracts/interfaces/IManageableVault.sol SHA3: ef185ae483f2673838b1337485b67991e66db730c4a3e968655052b6b56fcc7b File: contracts/interfaces/IPausable.sol SHA3: 29a7b5381d81dc05a2305f8db60b49147cd2b0d5ed470979f6221d01910199c4 File: contracts/interfaces/IRedemptionVault.sol SHA3: 76fd48d4e340fc8fa9bd6d240a601a0c2dc218e580fd18ee8d7271a9c2ca0d3a

	File: contracts/interfaces/IStUSD.sol SHA3: 5dcfd3e8b6df6844d7ecdb356cf0a86c6c652ba89ec88909c5ba918615545985 File: contracts/libraries/DecimalsCorrectionLibrary.sol SHA3: d8b962947c55546e35aa6491380d053949ea92952a3ad10d42ab798b64e60fb1

Second review scope

Repository	https://github.com/RedDuck-Software/midas-contracts
Commit	6079eb1b10efdfe99cebb4dc50dfb62c0fdf7d9c
Requirements	Midas doc
Contracts	File: DepositVault.sol SHA3: 4c2ac72bfd1cc07a30bf5a7aad50495fd50edf46dd6467b5dd32feb2ec9600d3 File: RedemptionVault.sol SHA3: 17c22ad04e5c70b15066c0cf24a64ba27ed37c80fbc9d7ec59fc29e68959f5ea File: stUSD.sol SHA3: f5d74b11a9876721c1b442a4e467f102b0f1d55215fb9dc9dc5e43a11d07f7ce File: abstract/ManageableVault.sol SHA3: 20f969f58c0797049c0e2f0a52abef9604fa7c74899327976cf96ad164c8d4dc File: access/Blacklistable.sol SHA3: 74c10f75f2208c7aa53bdf60b4e38ced375e14a0888e4a73e3f297642d012af2 File: access/Greenlistable.sol SHA3: c35afbf1c1daffec15c10128bb54ef64ad65c122fe3e64c0808b4ba3cedf01e0 File: access/MidasAccessControl.sol SHA3: 6942211c30e97fcb54c2271c6cefcd37d01544d0cb1b3cc236b6026cf148bc89 File: access/MidasAccessControlRoles.sol SHA3: 37c5b95da82cc9456720a49050b1cde4183f342371b4709e136862b0b35933c7 File: access/Pausable.sol SHA3: 883ad172649f54d18070fddd6a7ab74c49075e7af037542abce09e6266c79d03 File: access/WithMidasAccessControl.sol SHA3: 9dc0ed76ae9cd2777a954f1dc10e94d2f43a4169429249f3dcd8f17a4701a2a9 File: feeds/DataFeed.sol SHA3: 1727802dba54a5a482268c600d2e9e44545ad6704852042847e065f9363ebc71 File: interfaces/IDataFeed.sol SHA3: 961bb8991c85a68d86eb1811e81f8baaa1863bb2f7f1ea54ebd23d292d2eea5a File: interfaces/IDepositVault.sol SHA3: a5e3957985ed705e10f48b3b47f73495379c7053f3d8b2c9d7b7a2eb5239da7a File: interfaces/IManageableVault.sol SHA3: ef185ae483f2673838b1337485b67991e66db730c4a3e968655052b6b56fcc7b File: interfaces/IPausable.sol SHA3: 29a7b5381d81dc05a2305f8db60b49147cd2b0d5ed470979f6221d01910199c4 File: interfaces/IRedemptionVault.sol

	SHA3: 76fd48d4e340fc8fa9bd6d240a601a0c2dc218e580fd18ee8d7271a9c2ca0d3a File: interfaces/IStUSD.sol SHA3: 5dcfd3e8b6df6844d7ecdb356cf0a86c6c652ba89ec88909c5ba918615545985 File: libraries/DecimalsCorrectionLibrary.sol SHA3: d8b962947c55546e35aa6491380d053949ea92952a3ad10d42ab798b64e60fb1