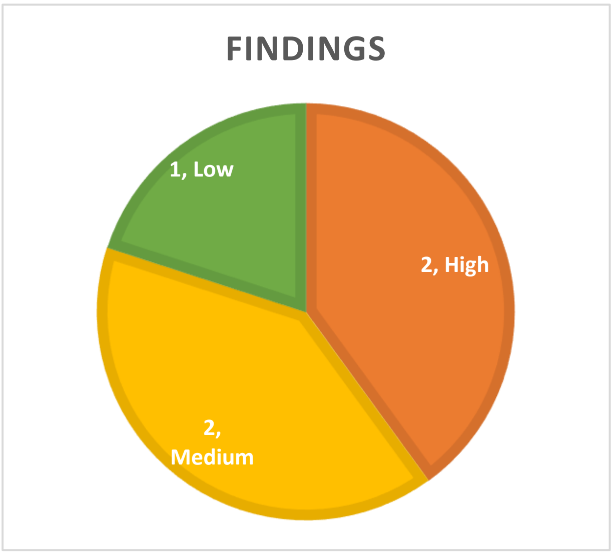
Trust Security

Smart Contract Audit

MegaStrategy

29/01/25

Executive summary

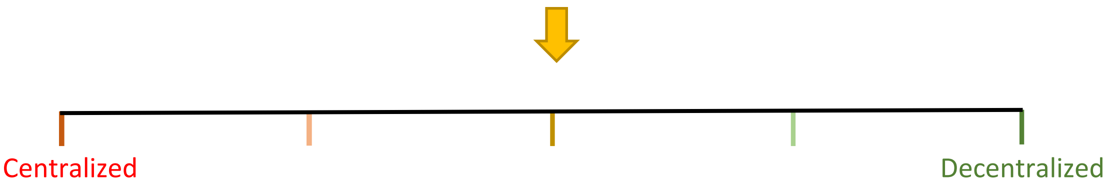


Category	Treasury Protocol
Audited file count	6
Lines of Code	666
Auditor	Trust
Time period	20/01/25-25/01/25

Findings

Severity	Total	Fixed	Acknowledged
High	2	2	-
Medium	2	1	1
Low	1	1	-

Centralization score



Signature

Digitally signed by Trust
Date: 2025.01.29 13:33:17 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	25/01/25	Client report
0.2	29/01/25	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- Banker.sol
- Issuer.sol
- MegaTokenOracle.sol
- TOKEN.v1.sol
- MegaToken.sol
- ConvertibleDebtToken.sol

Repository details

- **Repository URL:** https://github.com/MegaStrategy/megastrategy
- **Commit hash:** a76d72764b835d8ebc38edb0cdaa43f01b6e09ea
- **Fix commit hashes:**
    - **H-1** 6d503fd35a97e7914d839cc75de086f15c4054e8
    - **H-2** 90fb16c58d5bafa040fa48efe4c304e945037e61
    - **M-2** a13f108fc6bb7b58b97feddf0baa34a4fa3c5c36
    - **L-1** 8da1652f1af8e563eecca3a13dc2ebdec5195722
    - **R-1, R-4** 13f8863e974c2faa79b7d66f23a2574f968d86a1
    - **R-2, R-3** 2d894df3f54a0c87a4541a3cac7cdb8a269c9544

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time off auditing & bug hunting, he enjoys assessing bounty contests in C4 as a Supreme Court judge.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Project kept code as simple as possible, reducing attack risks
Documentation	**Excellent**	Project is mostly very well documented.
Best practices	**Excellent**	Project consistently adheres to industry standards.
Centralization risks	**Moderate**	Privileged users have significant impact on the safety and functionality of the platform.

Findings

High severity findings

TRST-H-1 All TOKENs backing non-exercised options will be stuck in the treasury

- **Category:** Integration issues
- **Source:** Issuer.sol
- **Status:** Fixed

Description

The issuer is tasked with creating and issuing OTokens, which can be traded in until the option expiry date. After that date, any TOKEN that has not been traded remains in the Teller contract, which expects the receiver of the OToken to call *reclaim()* to take in the non-traded units. However, the Issuer marks the TRSRY module as the receiver, which is a smart contract without supporting logic to call reclaim. As a result, the tokens would be stuck.

Recommended mitigation

Add the *reclaim()* functionality to the treasury, or use a different token as the receiver which would handle claiming.

Team response

Fixed.

Mitigation review

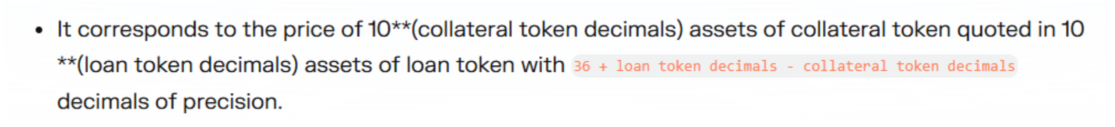
The issue has been addressed as suggested.

TRST-H-2 The integration with Morpho Blue Oracle exposes price with wrong decimals

- **Category:** Decimals issues
- **Source:** MegaTokenOracle.sol
- **Status:** Fixed

Description

Morpho supports arbitrary plug-in oracles. It defines the following specification for the *price()* function:


- It corresponds to the price of 10**(collateral token decimals) assets of collateral token quoted in 10 **(loan token decimals) assets of loan token with `36 + loan token decimals - collateral token decimals` decimals of precision.

The function has been coded in the Oracle contract below:

```
function price() external view returns (uint256) {
    // Scale: PRICE decimals
    // We know that PRICE decimals == TOKEN decimals == 18
    uint256 collateralPriceInLoanToken = PRICE.getPriceIn(TOKEN, loanToken);

    // Adjust to the expected scale
    return 1e36 * collateralPriceInLoanToken / _loanTokenScale;
}
```

The *PRICE.getPriceIn()* function returns the price of a decimals-aware unit of TOKEN quoted in decimals-aware unit of **loanToken**, scaled by 1e18. It can be assumed that the collateral token, which is TOKEN, has 18 decimals. Therefore, the Morpho calculation is **36 + loanDecimals – 18 = 18 + loanDecimals** decimals. Since the result is already in **18** decimals, it just needs to be scaled by **loanDecimals.** The current code has a completely different calculation, likely due to confusion from the required units.

Recommended mitigation

Use the scaling outlined above.

Team response

Fixed.

Mitigation review

Issue has been addressed as suggested.

Medium severity findings

TRST-M-1 The Debt token does not support various token price ratios

- **Category:** Precision loss issues
- **Source:** ConvertibleDebtToken.sol
- **Status:** Acknowledged

Description

Every ConvertibleDebtToken has a **conversionPrice**, defined as the amount of **underlying** per **convertedTo** token (decimals-aware), in **underlying** decimals. Suppose that underlying is a highly valued token, worth $101,000 per unit, with 6 decimals. Also, the convertedTo token (TOKEN) is worth $0.1 per unit, with 18 decimals. Therefore, the admin would like to input a **convertedTo** value of **$0.1/$101,000 * 1e6 < 1**. This means such a ratio cannot be expressed, as it would revert in the auction or any contract that uses the **conversionPrice**.

Recommended mitigation

The **conversionPrice** value should be scaled by 1e18 to support a wide variety of token value combinations.

Team response

We acknowledge this can happen with certain token decimals and price combinations. However, for the purposes of our usage, this is unlikely to happen because the underlying_token will always be a USD stablecoin. Therefore, the scenario of it having a small number of decimals (e.g. 6) AND a relative value of > 1e6 to the convertTo token is very low. We prefer to keep our price values consistent with the usage across other parts of the system. We have added a note of this possibility in the codebase of the token in case others decide to use it so they are aware of the limitation.

TRST-M-2 Conversion to TOKEN could fail in the Banker due to rounding errors

- **Category:** Rounding issues
- **Source:** Banker.sol
- **Status:** Fixed

Description

When issuing debt tokens, the Banker increases its mint approval amount so that it could later mint tokens during conversion. It is done in the lines below:

```
uint256 mintAmount = _getConvertedAmount(amount, conversionPrice);
TOKEN.increaseMintApproval(address(this), mintAmount);
```

```
function _getConvertedAmount(
    uint256 amount,
    uint256 conversionPrice
) internal view returns (uint256) {
    return (amount * 10 ** _tokenDecimals) / conversionPrice;
}
```

Note that the amount is rounded down. However, *issue()* can be called several times. The remainder of each call is always ignored, but the corresponding, non-dividing debt token amount is still minted. If those debt tokens remainders are added together, they would require additional units of TOKEN to be approved. Theoretically, this means *convert()* could fail as there is not enough approval from the individual *issue()* calls.

Recommended mitigation

Round up the mint approval to be added, or perform the approvals just before minting the tokens instead of during an *issue()*.

Team response

Fixed.

Mitigation review

Issue has been addressed by rounding up the approval.

Low severity findings

TRST-L-1 Upgrades of TRSRY and TOKEN modules lead to reverting behavior

- **Category:** Upgrade issues
- **Source:** Banker.sol, Issuer.sol
- **Status:** Fixed

Description

MegaStrategy leverages the Default Framework, which operates by installing and removing modules from the Kernel. At any point, the latest Module dependencies can be fetched through *configureDependencies(),* for example:

```
function configureDependencies() external override returns (Keycode[] memory
dependencies) {
    dependencies = new Keycode[](3);
    dependencies[0] = toKeycode("TRSRY");
    dependencies[1] = toKeycode("TOKEN");
    dependencies[2] = toKeycode("ROLES");

    TRSRY = TRSRYv1(getModuleAddress(dependencies[0]));
    TOKEN = TOKENv1(getModuleAddress(dependencies[1]));
    ROLES = ROLESv1(getModuleAddress(dependencies[2]));

    _tokenDecimals = TOKEN.decimals();
}
```

There are scenarios where modules are assumed to be the same at different points in the lifecycle of a contract, but due to a module upgrade, they refer to another contract. In the Banker, the TRSRY and TOKEN update their approval during *issue()*:

```
TRSRY.increaseWithdrawApproval(address(this), underlying, amount);

// Increase this contract's mint approval for the amount divided by the
conversion rate
// This is to ensure that the debt token can be converted
uint256 mintAmount = _getConvertedAmount(amount, conversionPrice);
TOKEN.increaseMintApproval(address(this), mintAmount);
```

Then those approvals are required during *convert()* or *redeem()*:

```
TRSRY.withdrawReserves(msg.sender, underlying, amount_);

// Calculate the amount of tokens that could have been minted against the
debt tokens
uint256 mintAmount = _getConvertedAmount(amount_, conversionPrice);

// Decrease the mint approval for the mint amount
// We do this since the debt token has been burned to avoid an extra
dangling mint allowance
TOKEN.decreaseMintApproval(address(this), mintAmount);
```

If one of these modules is upgraded, the functions may revert. A similar situation occurs in the Issuer. In *createO()*, TOKEN is registered as payout token. In *issueO()*, it will try to transfer the current TOKEN into the Teller:

```
// Approve the teller to pull the newly minted TOKENs
ERC20(address(TOKEN)).safeApprove(address(teller), amount_);

// Mint the oToken from the teller
teller.create(oToken(token_), amount_);
```

This would cause a mismatch, and the *issue()* would revert.

Recommended mitigation

Consider refactoring so that either a module upgrade does not cause any issues, or if a module upgrade should not be supported, freeze the current version into the contract and revert on any module changes.

Team response

Fixed.

Mitigation review

The issue has been addressed by (1) locking in place the TOKEN module, (2) acknowledging that a TRSRY upgrade would need to have migration logic to support the approvals made from previous Banker interactions.

Additional recommendations

TRST-R-1 Improve state changes in Banker

In *initialize()*, the Banker admin can change important configuration values and set the state to active. The function should only be called when the current state is inactive, but it is never validated.

TRST-R-2 Improve Documentation

The *onSettle()* function of Banker is incorrectly labelled as not implemented.

TRST-R-3 Defensive precautions in the Axis Auction integration

The Banker, which integrates with Axis Finance, is defined as its own Callback. Several improvements can be made:

- In the implemented hooks, ensure that the seller of the lot is **address(this)**, as done in *onCreate()*. It should never be otherwise.
- The *onPurchase()* and *onBid()* hooks should never be called, consider reverting in them to indicate they are not supported.

TRST-R-4 Check sanity in state changes

The *setConversionPrice()* function of ConvertibleDebtToken checks that the previous price is zero, since it should only be set once. However it should also be checked that the new value is not zero, otherwise the function executes successfully and emits and event, despite no changes taking place. Likewise, consider adding a check when activating and disactivating the Banker and Issuer modules, that the current state is the opposite of the new state.

TRST-R-5 Introduce permissioned burning functionality in TOKEN

Most modules in the Default Framework are designed so that it is easy to remove and install another one, i.e. that no high-complexity state is maintained in them. The MegaToken, however, maintains state on the balance, allowance, and voting checkpoints, of each user. This makes it difficult in the event of an emergency, to replace the token with a new one. Consider the possible remediations and alternatives. One way to de-risk the MegaToken is to allow a permissioned user to burn from other users (similar to minting). This way, if tokens are stuck, like in TRST-H-1 for example, they can be safely burnt without a complex redeploy.

Centralization risks

TRST-CR-1 The Default Framework is highly permissioned

There are various high-risk privileges in the different MegaStrategy modules. Those affect funds locked in the treasury as well as approvals to the modules. The owning Multisig and any delegated privileged address should be considered trusted.

Systemic risks

TRST-SR-1 External integrations are trusted

The contracts make use of several external integrations:

- LinearVesting contracts hold MegaToken options.
- Teller implements option minting and exercise logic.
- Axis Finance contracts implement the auction of debt tokens.

It should be clear that any issues with the contracts could compromise the security of MegaStrategy.

TRST-SR-2 Bad Debt risks

MegaStrategy issues debt tokens which are redeemable or convertible to Mega tokens. Holders of debt tokens should be aware of the possibility of default on the debt, as underlying is not contractually held in place in the treasury.