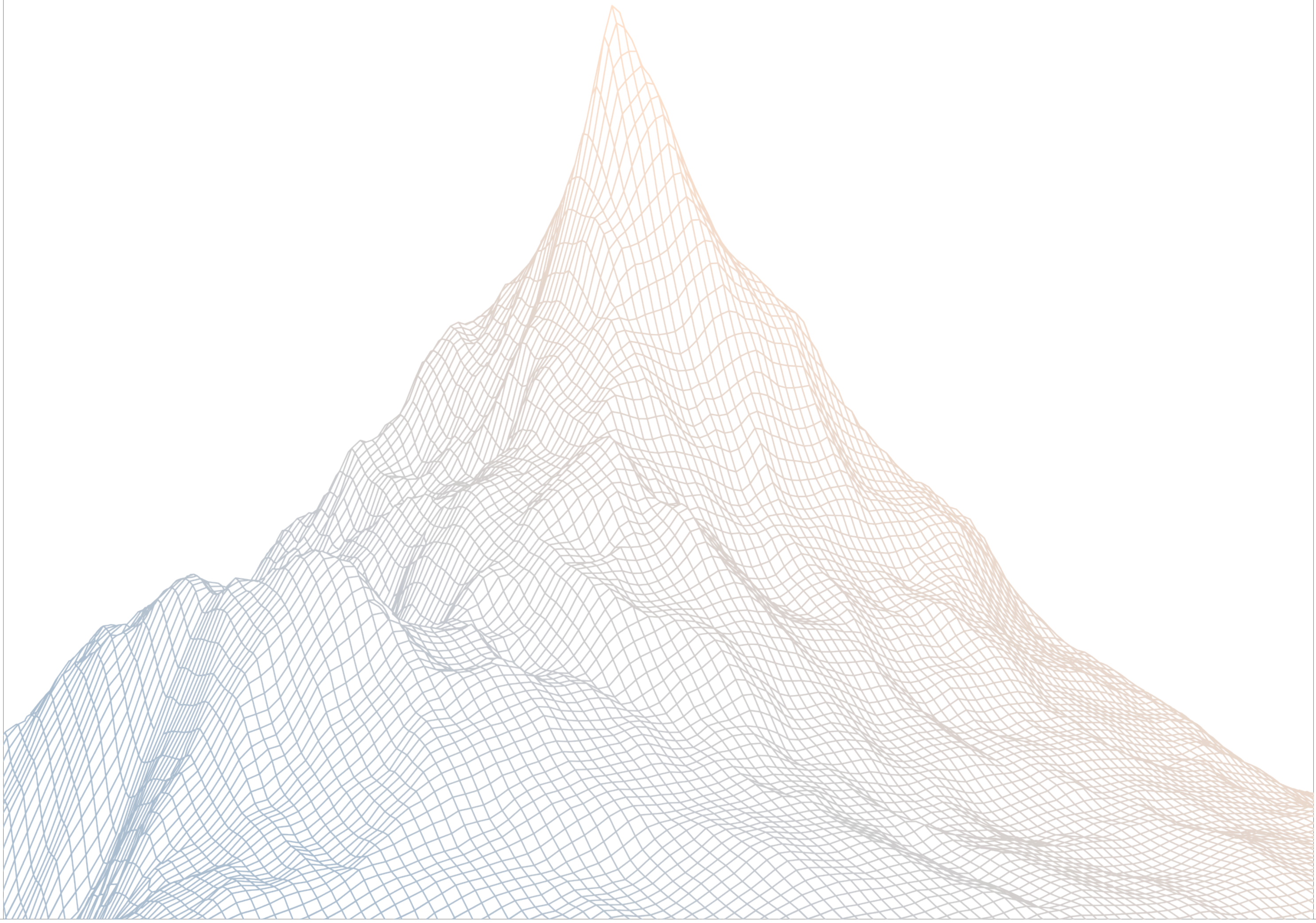
Zenith

Yala

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: March 17th to March 18th, 2025

AUDITED BY: 10xhash
said

Contents

1

Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About Yala

Yala is a liquidity layer for Bitcoin, unlocking Bitcoin liquidity and connecting it to cross-chain yield opportunities.

2.2 Scope

The engagement involved a review of the following targets:

Target	yala-core
Repository	https://github.com/yalaorg/yala-core
Commit Hash	19f227ebc31a2c6cb23bb5492d5b9a5f2160caa6
Files	crsm/CRSM.sol crsm/CRSMFactory.sol oft/DebtTokenOFT.sol

2.3 Audit Timeline

March 17, 2025	Audit start
March 18, 2025	Audit end
March 21, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	4
Informational	1
Total Issues	**7**

3

Findings Summary

ID	Description	Status
M-1	DEBT_GAS_COMPENSATION can be instantly updated causing users to suffer losses	Resolved
M-2	DEBT_GAS_COMPENSATION can be instantly updated causing users to suffer losses	Resolved
L-1	strict TARCR check inside the repay operation might cause an issue.	Resolved
L-2	Token Id Front-Running Potential Issue	Resolved
L-3	Not considering stability pool yield and debt token balance inside CSRM during the repay operation	Resolved
L-4	Excess assets can be withdrawn from the StabilityPool due to not considering minNetDebt	Resolved
I-1	CRSM ownership is not tied to the minted ERC721 within the factory.	Resolved

4 Findings

4.1 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] DEBT_GAS_COMPENSATION can be instantly updated causing users to suffer losses

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- CRSM.sol

Description:

Users are incentivized to invoke `repay` as `DEBT_GAS_COMPENSATION` is paid as reward. But the owner of CRSM contract can update this value instantly thereby changing the reward received by the invoker

```
function setDebtGasCompensation(uint256 _debtGasCompensation)
    external onlyOwner {
    DEBT_GAS_COMPENSATION = _debtGasCompensation;
    emit DebtGasCompensationUpdated(_debtGasCompensation);
}
```

Eg: User spends 10$ in gas to invoke repay since currently DEBT_GAS_COMPENSATION is 20 Owner front-runs and sets DEBT_GAS_COMPENSATION to 0. Now the user will suffer a loss and owner can benefit from paying a lower gas amount (compared to himself invoking repay)

Recommendations:

Enforce a time delay on the update

Yala: Resolved with @9fbaa62fa8...

Zenith: Verified.

[M-2] DEBT_GAS_COMPENSATION can be instantly updated causing users to suffer losses

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- CRSM.sol

Description:

Users are incentivized to invoke `repay` as `DEBT_GAS_COMPENSATION` is paid as reward. But the owner of CRSM contract can update this value instantly thereby changing the reward received by the invoker

```
function setDebtGasCompensation(uint256 _debtGasCompensation)
    external onlyOwner {
    DEBT_GAS_COMPENSATION = _debtGasCompensation;
    emit DebtGasCompensationUpdated(_debtGasCompensation);
}
```

Eg: User spends 10$ in gas to invoke repay since currently DEBT_GAS_COMPENSATION is 20 Owner front-runs and sets DEBT_GAS_COMPENSATION to 0. Now the user will suffer a loss and owner can benefit from paying a lower gas amount (compared to himself invoking repay)

Recommendations:

Enforce a time delay on the update

Yala: Resolved with @9fbaa62fa8...

Zenith: Verified.

4.2 Low Risk

A total of 4 low risk findings were identified.

[L-1] strict `TARCR` check inside the `repay` operation might cause an issue.

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- CRSM.sol#L63

Description:

Inside `repay` operation, it will always check that the new ICR must always greater than `TARCR`.

```
    function repay(uint256 amount) external {
        ITroveManager.Trove memory trove
    = troveManager.getCurrentTrove(troveId);
        uint256 entireDebt = trove.debt + trove.interest;
        require(amount <= entireDebt, "CRSM: Too much debt to repay");
        uint256 price = troveManager.fetchPrice();
        uint256 ICR = YalaMath._computeCR(trove.coll, entireDebt, price);
        require(ICR <= TRGCR, "CRSM: ICR must be below TRGCR");
        uint256 deposits
    = stabilityPool.getCompoundedDebtDeposit(address(this));
        require(amount + DEBT_GAS_COMPENSATION <= deposits, "CRSM:
    Insufficient deposits");
        stabilityPool.withdrawFromSP(amount + DEBT_GAS_COMPENSATION);
        borrowerOperations.repay(troveManager, troveId, amount);
        trove = troveManager.getCurrentTrove(troveId);
        uint256 newICR = YalaMath._computeCR(trove.coll, trove.debt
    + trove.interest, price);
>>>  require(newICR >= TARCR, "CRSM: ICR ≥ TARCR");
        require(newICR <= MAX_TARCR, "CRSM: ICR ≤ MAX_TARCR");
        IERC20(debtToken).safeTransfer(msg.sender, DEBT_GAS_COMPENSATION);
        emit Repay(troveManager, troveId, amount);
    }
```

This could cause an issue in the case where all compounded debt deposits inside the stability pool have already been withdrawn, but the `TARCR` still cannot be reached. This prevents the use of debt tokens inside the stability pool to improve the trove's ICR.

Recommendations:

consider checking `newICR` against `TARCR` only if the repay operation does not use all compounded debt deposits.

Yala: Resolved with @d422097fc23...

Zenith: Verified.

[L-2] Token Id Front-Running Potential Issue

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- CRSMFactory.sol#L31-L46

Description:

When creating new CRSM inside factory, the `crsm` address salt only based on `tokenId`.

```
    function createNewCRSM(ITroveManager troveManager, uint256 troveId,
    uint256 _TRGCR, uint256 _TARCR, uint256 _MAX_TARCR,
    uint256 _debtGasCompensation, uint256 _amount)
    external returns (ICRSM crsm) {
        address collateralToken
    = address(borrowerOperations.collateraTokens(troveManager));
        require(collateralToken != address(0), "CRSMFactory: nonexistent
    TM");
        address owner = troveManager.ownerOf(troveId);
        require(msg.sender == owner, "CRSMFactory: not trove owner");
        uint256 tokenId = nonce++;
        address implementation = crsmImpl;
>>>  crsm = ICRSM(implementation.cloneDeterministic(bytes32(tokenId)));
        crsm.setParameters(troveManager, troveId, owner, _TRGCR, _TARCR,
    _MAX_TARCR, _debtGasCompensation);
        crsms[tokenId] = crsm;
        _mint(owner, tokenId);
        if (_amount > 0) {
            deposit(crsm, _amount);
        }
        emit NewCRSMDeployment(troveManager, troveId, tokenId, crsm);
    }
```

This is open front-running attack vector :

1. Alice initiates a transaction to create a new CRSM with `tokenId` N and and create separate deposit tx to the pre-calculated `crsm` address using `tokenId` N.

2. Bob sees this pending transaction
3. Bob front-run and create the CRSM with the `tokenId` N
4. Alice deposit is executed and wrongly send the debt token to the bob's `crsm`.

Recommendations:

Consider also incorporating `msg.sender` for the salt.

Yala: Resolved with @5e7c7f4665...

Zenith: Verified

[L-3] Not considering stability pool yield and debt token balance inside CSRM during the `repay` operation

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Resolved	`LIKELIHOOD:` Medium

Target

- CRSM.sol#L50-L67

Description:

When the `repay` operation is called, it will attempt to withdraw debt tokens from the stability pool and use them to repay the debt. However, it does not consider the potential debt tokens accrued from the stability pool or, in edge cases, the debt token balance inside the CSRM.

Not accounting for these factors could cause issue for trove position in cases where the compounded debt deposited in the stability pool is insufficient to cover the repayment operation, preventing the `ICR` from reaching the `TARCR`.

Recommendations:

Consider to utilize accrued yield and debt token balance inside the CSRM when `repay` operation is called.

Yala: Resolved with @25e6858a9db...

Zenith: Verified

[L-4] Excess assets can be withdrawn from the StabilityPool due to not considering minNetDebt

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- CRSM.sol

Description:

The repay function allows for amount to be `trove.debt + trove.interest`. But since actual repayment function limits the repayable debt amount to `trove.debt - minNetDebt`. This can cause excess tokens to be withdrawn from the stability pool and left idle in the CSRM contract

```
function repay(uint256 amount) external {
    ITroveManager.Trove memory trove
    = troveManager.getCurrentTrove(troveId);
    uint256 entireDebt = trove.debt + trove.interest;
    require(amount <= entireDebt, "CRSM: Too much debt to repay");
    uint256 price = troveManager.fetchPrice();
    uint256 ICR = YalaMath._computeCR(trove.coll, entireDebt, price);
    require(ICR <= TRGCR, "CRSM: ICR must be below TRGCR");
    uint256 deposits
    = stabilityPool.getCompoundedDebtDeposit(address(this));
    require(amount + DEBT_GAS_COMPENSATION <= deposits, "CRSM: Insufficient
    deposits");
    stabilityPool.withdrawFromSP(amount + DEBT_GAS_COMPENSATION);
    borrowerOperations.repay(troveManager, troveId, amount);
```

```
(vars.collChange, vars.isCollIncrease) = _getCollChange(_collDeposit,
    _collWithdrawal);
if (!_isDebtIncrease && _debtChange > 0) {
    if (_debtChange > (vars.debt - minNetDebt)) {
        vars.debtChange = vars.debt - minNetDebt;
        _debtChange = _debtChange - vars.debtChange;
        vars.interestRepayment = YalaMath._min(_debtChange, vars.interest);
```

Recommendations:

Only allow repayment of `trove.debt + trove.interest - minNetDebt` amount

Yala: Resolved with @701be9f50e...

Zenith: Verified

4.3 Informational

A total of 1 informational findings were identified.

[I-1] CRSM ownership is not tied to the minted ERC721 within the factory.

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- CRSM.sol
- CRSMFactory.sol#L62-L66

Description:

When a new CRSM is created via the factory contract, a new ERC721 token is minted to the owner. However, the ownership of this ERC721 token and the ownership of the `crsm` contract (implemented using OpenZeppelin's `Ownable`) are not tied together. This means a user can transfer ownership of the crsm contract or transfer ownership of the ERC721 token without affecting the other.

Recommendations:

Consider designing it more clearly by either removing the ERC721 token functionality inside the factory or adjusting the crsm access control to use ERC721 ownership.

Yala: Resolved with @efa364342d7...

Zenith: Verified.