MUSTANG

SECURITY AUDIT REPORT

PREPARED BY: SHRED SECURITY

30 Nov 2025

Security Researchers
0xrex | yashar | kenzo

Table of Contents

About Shred Security

Shred Security provides high quality security audits for blockchain and DeFi protocols across different chains. Our audits consistently uncover high-impact vulnerabilities missed by others, backed by a proven track record of top competition placements and security partnerships with leading protocols.

Learn more about us: shredsec.xyz

Protocol Executive Summary

Must Finance is a liquity V2 fork for the Saga EVM, with additional collaterals and security features. In this review, our researchers have audited the TroveId.sol, BatchId.sol and LiquityMath.sol. As the contracts will be deployed on SAGA. The SAGA EVM is on an older version of EVM, so the Liquity V2 files were compiled with Shanghai and moving the contracts back to solidity 0.8.23 from 0.8.24. This audit was performed to check if any security issues occur on the given files during this compiler changes from 0.8.24 to 0.8.23

Disclaimer

The Shred Security team makes all effort to find as many vulnerabilities in the code in the given time period. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and this report neither endorses any specific project or team nor assures the complete security of the project.

Risk Classification

Likelihood \ Impact	High	Medium	Low
High	High	High/Medium	Medium
Medium	High/Medium	Medium	Medium/Low
Low	Medium	Medium/Low	Low

Executive Summary

The shred security team has conducted the review for 1 day in total. In this period of time, a total of 2 issues were found.

About the Project

Field	Value
Project Name	Must Finance Contracts
Repository	https://github.com/MustangProtocol/must-finance/commit/399c296bcd221a64b695a86f9d70a71f7dbedf40
Commit	7dbedf40
Type of Project	DeFi Protocol
Lines of Code	100

Audit Timeline

Phase	Date
Audit Start	10/11/2025
Audit End	11/11/2025
Report Published	30/11/2025

Vulnerability Summary

Severity	Count	Fixed	Acknowledged
High Risk	1		1
Medium Risk	1		1
Low Risk	0		
Informational	0		
Total	**2**		**2**

Findings Summary

Issue ID	Description	Severity	Status
H-1	Unsupported WETH calls break core functionality	High	Acknowledged
M-1	swapToBold calls would fail during zapper trove operations	Medium	Acknowledged

Findings

High Severity

[H-01] Unsupported WETH calls break core functionality

Impact: High
Likelihood: High

Description

The Saga "WETH" contract is a standard ERC-20 token and does not implement `deposit()` or `withdraw()` functions.
However, multiple contracts within the Must codebase assume Ethereum-style WETH behavior and attempt to convert between ETH and WETH using these functions.

Examples include:

```
// Convert ETH to WETH
WETH.deposit{value: msg.value}();
```

```
// Convert WETH to ETH
WETH.withdraw(_amount);
(bool success,) = receiver.call{value: _amount}("");
```

Functions making calls to WETH deposit or withdraw:

- `LeverageWETHZapper.sol::openLeveragedTroveWithRawETH()`
- `LeverageLSTZapper.sol::openLeveragedTroveWithRawETH()`
- `WETHZapper.sol::openTroveWithRawETH()`
- `WETHZapper.sol::addCollWithRawETH()`
- `WETHZapper.sol::_adjustTrovePre()`
- `WETHZapper.sol::withdrawCollToRawETH()`
- `WETHZapper.sol::_adjustTrovePost()`
- `WETHZapper.sol::closeTroveToRawETH()`
- `WETHZapper.sol::receiveFlashLoanOnCloseTroveFromCollateral()`

All of these functions will revert on Saga, since the WETH contract lacks the required functions, and ETH is not the native gas token.

Recommendation

Remove all calls to `WETH.deposit` and `WETH.withdraw`.

Client Response: Acknowledged

Medium Severity

[M-01] `swapToBold` calls would fail during zapper trove operations

Impact: Medium
Likelihood: Medium

Description

The zappers open up ways for users to zap into and out of debt positions by also offering leveraging opportunities. This critically makes use of Dex Pools (Exchange Modules such as Curve, Uniswap etc.) to seed extra liquidity/collateral as well as swap from the collateral tokens to $MUST during debt positions closure to have enough $MUST tokens available during the call to `borrowerOperations.closeTrove()`.

However, since we commentted out the approval logic of the collateral token to the exchange in the constructors of inherited GasCompZapper and WETHZapper constructors, these swap calls would fail as we can see from taking a look at the Exchange Module's `swapToBold()` function below:

```
function swapToBold(uint256 _collAmount, uint256 _minBoldAmount)
external returns (uint256) {
    ICurvePool curvePoolCached = curvePool;
    uint256 initialCollBalance =
collToken.balanceOf(address(this));
    // @note requires approval from sender prior
    collToken.safeTransferFrom(msg.sender, address(this),
_collAmount);
    collToken.approve(address(curvePoolCached), _collAmount);
    ...
}
```

And evidently in the `exchange.swapToBold()` function below, this would revert since approvals to the exchange will be missing for the collateral token.

```
function receiveFlashLoanOnCloseTroveFromCollateral(
    CloseTroveParams calldata _params,
    uint256 _effectiveFlashLoanAmount
) external {
    require(msg.sender == address(flashLoanProvider), "GCZ: Caller
not FlashLoan provider");

    LatestTroveData memory trove =
troveManager.getLatestTroveData(_params.troveId);
    uint256 collLeft = trove.entireColl - _params.flashLoanAmount;
    require(collLeft >= _params.minExpectedCollateral, "GCZ: Not
enough collateral received");

    // @audit [M] Will always revert during exchange.swapToBold()
call
    exchange.swapToBold(_effectiveFlashLoanAmount,
trove.entireDebt);

    borrowerOperations.closeTrove(_params.troveId);

    collToken.safeTransfer(address(flashLoanProvider),
_params.flashLoanAmount);
    collToken.safeTransfer(_params.receiver, collLeft);
}
```

Recommendation

If the intention is to disallow the exchange modules (Curve, Uniswap etc) from trove opening and closing operations while utilizing the zappers, then it is recommended to omit the `exchange.swapToBold` and `exchange.swapFromBold` calls. Otherwise, include the `collToken.approve(address(_exchange), type(uint256).max);` function calls to allow zapping into positions utilizing flashloans in the zapper contracts.

This issue is evident in the `GasCompZapper::constructor` and `WETHZapper::constructor` thus affecting the following contract calls during flashloan operations for trove debt management:

1. LeverageLSTZapper::receiveFlashLoanOnLeverDownTrove
2. WETHZapper::receiveFlashLoanOnCloseTroveFromCollateral
3. WrappedTokenZapper::receiveFlashLoanOnCloseTroveFromCollateral
4. LeverageWETHZapper::receiveFlashLoanOnLeverDownTrove

Client Response: Acknowledged

Appendix

The reviewed files — TroveId, BatchId, and LiquityMath—were extensively tested against Solidity 0.8.23. No major overflow or underflow issues were found. Cancun upgrade changes do not affect current implementations. We are providing our fuzzing test suite here

Overflow/Underflow behaviors were validated in multiple edge cases, confirming that functions revert safely under extreme inputs.

decMul overflow

`LiquityMath.decMul()` does `x * y` first, then divides. If the product overflows uint256, it reverts.

Happens in `_decPow()` when squaring large base values.

```
function testEdgeCase_DecMulOverflow() public {
    uint256 largeX = 2**128;
    uint256 largeY = 2**128;

    bool wouldOverflow = largeX > 0 && largeY > type(uint256).max
/ largeX;
    assertTrue(wouldOverflow);

    try this.callDecMul(largeX, largeY) returns (uint256) {
        revert("should overflow");
    } catch {
        // expected
    }
}
```

_decPow internal overflow

Even with the 525600000 minute cap, `decMul()` calls inside can still overflow when:

- Base is near `DECIMAL_PRECISION * 2`
- Squaring operations blow up

```
function testEdgeCase_DecPowOverflow() public {
    uint256 largeBase = DECIMAL_PRECISION * 2 - 1;
    uint256 exponent = 1000;

    try mathTester.callDecPow(largeBase, exponent) returns
(uint256 result) {
```

```
        assertGe(result, 0);
    } catch {
        // overflow happens sometimes
    }
}
```

_computeCR overflow

Does `_coll * _price`. Overflows if that product exceeds uint256 max.

```
function testEdgeCase_ComputeCROverflow() public {
    uint256 largeColl = 2**128;
    uint256 largePrice = 2**128;
    uint256 debt = 1e18;

    bool wouldOverflow = largeColl > 0 && largePrice >
type(uint256).max / largeColl;
    assertTrue(wouldOverflow);

    try this.callComputeCR(largeColl, debt, largePrice) returns
(uint256) {
        revert("should overflow");
    } catch {
        // expected
    }
}
```