RateHopper 2

SECURITY AUDIT REPORT

3 Apr 2026

Prepared by: **Shred Security**

Security Researchers
Oxrex - 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

Ratehopper is an intelligent DeFi platform that helps you find the best lending and borrowing rates across multiple protocols and automate position switching to maximize your returns. In addition, we have different strategies that allow you to deploy different investing strategies based on your risk appetite.

The team introduced enhanced collateral withdrawal logic (PR #6) that enables safe partial debt repayment scenarios where full debt cannot be cleared, leaving residual debt on the protocol. This update includes on-chain calculation of the maximum withdrawable collateral amount, support for type(uint256).max as a valid collateralAsset parameter, new Aave and Morpho oracle interfaces for accurate asset pricing, protocol handler enhancements across Aave V3, Compound, Morpho, Fluid, and Moonwell for max-withdrawal validation, refactored Fluid exit config, Moonwell ETH balance tracking (with WETH standardization), filename consistency fixes, and expanded test coverage with debug logging to improve flexibility in debt swaps and user experience during partial position exits.

The (PR #7) introduces a fix for Morpho integrations that resolves approval-related transaction failures during switchFrom operations. This update includes adding a buffer to the approve amount in the MorphoHandler to account for share-based repayment discrepancies (where the actual repay amount can exceed the approved amount by 1 unit), automatic approval reset to zero after the operation, and reinstatement of this Morpho-specific safety logic to prevent reverts and ensure robust position switching across the protocol.

Disclaimer

A smart contract security review cannot guarantee the complete elimination of vulnerabilities. The process is limited by time, available resources, and human expertise, and is intended to identify as many potential issues as reasonably possible. As such, no assurance can be given that all vulnerabilities will be discovered, or that the reviewed smart contracts are entirely secure. To strengthen security over time, follow-up audits, bug bounty programs, and continuous on-chain monitoring are strongly recommended.

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 3 days in total. In this period of time, a total of 9 issues were found.

About the Project

Project Name	Ratehopper Contracts
Repository	https://github.com/RateHopper/ratehopper-contracts
Commit	64294a0
Type of Project	Intelligent DeFi platform
Lines of Code	300

Audit Timeline

Audit Start	20/03/2026
Audit End	23/03/2026
Report Published	03/04/2026

Vulnerability Summary

Severity	Count	Fixed	Acknowledged
High Risk	0		
Medium Risk	4	4	
Low Risk	2	2	
Informational	3	1	2
Total	9	7	2

Findings Summary

Issue ID	Description	Severity	Status
M-1	`withdraw()` function of the MorphoHandler allows withdrawal of non-whitelisted assets	Medium	Fixed
M-2	AaveV3Handler._calculateMaxWithdrawAmount overestimates limits due to LT drift	Medium	Fixed
M-3	CompoundHandler._calculateMaxWithdrawAmount underestimates limits for multi-collateral positions	Medium	Fixed
M-4	`MoonwellHandler.getDebtAmount` Uses Stale `borrowBalanceStored` — Full Debt Repayment Leaves Residual Debt	Medium	Fixed
L-1	`CompoundHandler.supply` and `withdraw` accept arbitrary Comet address via `extraData`	Low	Fixed
L-2	MoonwellHandler pulls entire Safe balance instead of redeemed amount	Low	Fixed
I-1	Unsafe `IERC20.transfer` usage instead of `SafeERC20.safeTransfer`	Informational	Fixed
I-2	Inconsistent error handling: mix of `require` strings and custom errors	Informational	Acknowledged
I-3	Hardcoded interest rate mode `2` (variable) in `AaveV3Handler`	Informational	Acknowledged

Findings

Medium Severity

[M-1] `withdraw()` function of the MorphoHandler allows withdrawal of non-whitelisted assets

Affected File: contracts/protocols/MorphoHandler.sol
Severity: Medium
Impact: Medium
Likelihood: Medium

Description

```
function withdraw(address asset, uint256 amount, address onBehalfOf, bytes calldata ex
traData) external override onlyAuthorizedCaller(onBehalfOf) {
@>        require(registry.isWhitelisted(asset), "Asset is not whitelisted");

        // Decode market parameters from extraData
@>        (MarketParams memory marketParams, ) = abi.decode(extraData, (MarketParams,
 uint256));

        uint256 withdrawAmount = amount;
        if (amount == type(uint256).max) {
            withdrawAmount = _calculateMaxWithdrawAmount(marketParams, onBehalfOf);
            require(withdrawAmount > 0, "No collateral available to withdraw");
        }

        // Withdraw collateral from user's position to this contract
        morpho.withdrawCollateral(marketParams, withdrawAmount, onBehalfOf, address(th
is));
    }
```

In the above line, we want to ensure that non-whitelisted assets cannot be withdrawn thus, we check that the `asset` argument address is indeed in the whitelisted assets of the registry but the problem is that the `asset` is not bounded to `marketParams.collateral`. Below is the MarketParams struct definition:

```
struct MarketParams {
    address loanToken;
    address collateralToken; // @note token being withdrawn
    address oracle;
    address irm;
    uint256 lltv;
}
```

So, let's assume we have a state whereby:

- WETH is no longer whitelisted. USDC still is.
- User can provide USDC as `asset` but then encode WETH inside `extraData.collateralToken`. Thus, what happens then is that the user has breached the whitelist check enforced in `require(registry.isWhitelisted(asset), "Asset is not whitelisted");` to now withdraw non-whitelisted asset WETH by providing a whitelisted USDC as `asset` and WETH encoded in the extraData.

Impact

The whitelist can be breached in the scenario described above since the `asset` is not bounded to the `collateralToken` in the MarketParams field encoded data.

Recommendation

```
 function withdraw(address asset, uint256 amount, address onBehalfOf, bytes calldata e
xtraData) external override onlyAuthorizedCaller(onBehalfOf) {
-       require(registry.isWhitelisted(asset), "Asset is not whitelisted");

         // Decode market parameters from extraData
        (MarketParams memory marketParams, ) = abi.decode(extraData, (MarketParams, ui
nt256));
+       require(registry.isWhitelisted(marketParams.collateralToken), "Asset is not wh
itelisted");
 }
```

[M-2] AaveV3Handler._calculateMaxWithdrawAmount overestimates limits due to LT drift

Severity: Medium
Affected Contracts: `AaveV3Handler.sol`
Affected Functions: `_calculateMaxWithdrawAmount`

Summary

The `AaveV3Handler._calculateMaxWithdrawAmount` function incorrectly utilizes the portfolio's aggregate Liquidation Threshold (LT) to calculate the maximum safe withdrawal amount. Because withdrawing a specific asset alters the weighted average LT of the remaining portfolio, the handler's calculation consistently overestimates the amount that can be safely withdrawn. This results in transaction reverts within Aave's `ValidationLogic`.

Technical Details

Aave V3 calculates a user's Health Factor (HF) based on the weighted average Liquidation Threshold of all supplied collateral.

```
HF = (Σ (Collateral_i * Price_i * LT_i)) / (TotalDebtBase * 10,000)
```

The current implementation in `AaveV3Handler` attempts to calculate the maximum withdrawal by assuming the aggregate LT (`currentLiquidationThreshold`) remains constant:

```
// AaveV3Handler.sol L220-228
uint256 minCollateralBase = (totalDebtBase * LIQ_THRESHOLD_PRECISION) / currentLiquida
tionThreshold;
uint256 maxWithdrawBase = totalCollateralBase - minCollateralBase;
```

The Flaw:
When an asset is withdrawn, the `currentLiquidationThreshold` is immediately re-weighted. If a user withdraws an asset with an LT higher than the current portfolio average, the new average LT drops. By using the *old* average to calculate the withdrawal, the handler fails to account for this "LT drift," leading to an overestimated withdrawal amount that would push the HF below 1.0.

Impact

This logic error leads to immediate transaction reverts by Aave's core protocol when a user attempts a full withdrawal (`type(uint256).max`) of a high-LT asset. While Aave's internal validation prevents actual liquidation in this scenario, the flaw effectively breaks the "max withdraw" functionality for multi-collateral positions and results in a poor user experience.

Recommendation

The calculation should be refactored to use the specific Liquidation Threshold of the asset being extracted. This ensures the withdrawal limit accurately reflects how much "weighted backing" is being removed from the portfolio.

```
+ (, , uint256 assetLT, , , , , , , ) = dataProvider.getReserveConfigurationData(asse
t);
+ uint256 currentWeightedCollateralBase = (totalCollateralBase * currentLiquidationThr
eshold) / LIQ_THRESHOLD_PRECISION;
+
+ if (currentWeightedCollateralBase <= totalDebtBase) return 0;
+
+ uint256 maxWithdrawBase = ((currentWeightedCollateralBase - totalDebtBase) * LIQ_THR
ESHOLD_PRECISION) / assetLT;
```

[M-3] CompoundHandler._calculateMaxWithdrawAmount underestimates limits for multi-collateral positions

Severity: Medium
Affected Contracts: `CompoundHandler.sol`
Affected Functions: `_calculateMaxWithdrawAmount`

Summary

The `_calculateMaxWithdrawAmount` function in `CompoundHandler.sol` incorrectly assumes that the asset being withdrawn must collateralize the user's entire borrow balance. It fails to account for other collateral assets the user may have supplied to the same Compound V3 (Comet) instance. This results in an overly conservative withdrawal limit, potentially blocking safe withdrawals when using `type(uint256).max`.

Technical Details

In Compound V3, a user's borrowing capacity is determined by the aggregate value of all supplied collateral, each weighted by its specific `borrowCollateralFactor`.

The current implementation of `_calculateMaxWithdrawAmount` isolates the calculation to the target asset:

```
// CompoundHandler.sol L197-199
uint256 borrowValueBase = (borrowBalance * basePrice + baseScaleVal - 1) / baseScaleVa
l;
uint256 minCollateralValue = (borrowValueBase * 1e18 + uint256(assetInfo.borrowCollate
ralFactor) - 1) / uint256(assetInfo.borrowCollateralFactor);
uint256 minCollateral = (minCollateralValue * uint256(assetInfo.scale) + collateralPri
ce - 1) / collateralPrice;
```

This logic calculates the amount of *this specific asset* required to back the *entire* `borrowBalance`. If a user has other collateral (e.g., ETH) backing their USDC borrow and tries to withdraw a second collateral asset (e.g., WBTC), the function will return a lower-than-actual limit because it ignores the ETH's contribution to the borrowing capacity.

Impact

This is a functionality limitation that forces `withdraw(type(uint256).max)` to return a sub-optimal amount (often 0) for users with multi-asset collateral positions. It does not present a risk to funds since the error is in the conservative direction and Compound's core protocol would revert any truly unsafe withdrawal regardless of the handler's calculation.

The impact is further mitigated because `switchFrom` and full `exit` flows either repay debt first or use actual balances rather than this calculated limit.

Recommendation

Update the function to factor in the total collateral value across all assets supported by the Comet instance, or mirror Compound's internal `isBorrowCollateralized` logic to determine the safe withdrawal amount more accurately.

[M-4] `MoonwellHandler.getDebtAmount` Uses Stale `borrowBalanceStored` — Full Debt Repayment Leaves Residual Debt

Severity: Medium
Affected Contracts: `MoonwellHandler.sol`
Affected Functions: `getDebtAmount`

Summary

The `MoonwellHandler.getDebtAmount` function utilizes `borrowBalanceStored()` to query a user's debt. In Moonwell (a Compound V2 fork), `borrowBalanceStored()` returns the balance at the last interest accrual checkpoint without triggering a new accrual. This leads to an underestimation of the actual debt, causing incomplete repayments during full debt swaps or exits.

Technical Details

Moonwell provides two methods for querying borrow balances:

1. `borrowBalanceStored(address)` : Returns the balance stored in the last accrued block.
2. `borrowBalanceCurrent(address)` : Triggers `accrueInterest()` and then returns the up-to-date balance.

The current implementation of `getDebtAmount` uses the stale version:

```
// MoonwellHandler.sol L107
return IMToken(mContract).borrowBalanceStored(onBehalfOf);
```

When a user performs a "full" debt swap or exit using `type(uint256).max`, the protocol relies on `getDebtAmount` to determine the swap/repay size. Because the

interest for the current period is not included, the flash loan (in `executeDebtSwap`) or the transfer (in `exit`) is sized for the stale amount.

During the subsequent `repayBorrowBehalf` call, Moonwell internally accrues interest. Since the repayment amount was calculated based on the stale balance, a residual debt equal to the newly accrued interest remains on the protocol. This is inconsistent with the `repay` function in the same contract, which correctly uses `borrowBalanceCurrent` (L354).

Impact

Users attempting to fully migrate or close their Moonwell positions will unknowingly leave behind residual debt. This "dust" debt continues to accrue interest and may eventually push a position toward liquidation or require manual intervention to fully clear.

Recommendation

Update `getDebtAmount` to use `borrowBalanceCurrent()`. Note that this requires changing the function signature from `view` to non-attribute (or `payable` if applicable) in the interface, as `borrowBalanceCurrent` modifies state.

```
- function getDebtAmount(address asset, address onBehalfOf, bytes calldata) external v
iew returns (uint256) {
+ function getDebtAmount(address asset, address onBehalfOf, bytes calldata) external r
eturns (uint256) {
      address mContract = getMContract(asset);
      if (mContract == address(0)) revert TokenNotRegistered();
-     return IMToken(mContract).borrowBalanceStored(onBehalfOf);
+     return IMToken(mContract).borrowBalanceCurrent(onBehalfOf);
  }
```

Alternatively, apply a small buffer (e.g., 0.05%) to the stored balance to cover expected interest accrual, similar to the implementation in other handlers.

Low Severity

[L-1] `CompoundHandler.supply` and `withdraw` accept arbitrary Comet address via `extraData`

File: CompoundHandler.sol and L157-L179

Description: In `supply()` and `withdraw()`, the Comet contract address is decoded directly from user-supplied `extraData` rather than looked up via `getCContract(asset)` (which queries the registry). While the asset itself is whitelist-

checked, the Comet address is not validated against the registry. A malicious or misconfigured `extraData` could point to an arbitrary contract.

```
// supply() - cContract from extraData
address cContract = abi.decode(extraData, (address));
require(cContract != address(0), "Invalid comet address");

// Other functions like borrow(), repay() use getCContract(asset) from registry
address cContract = getCContract(asset); // safe lookup
```

Recommendation: Validate `cContract` against the registry mapping: `require(cContract == getCContract(asset), "Comet address mismatch")`, or always use `getCContract(asset)` for consistency.

[L-2] MoonwellHandler pulls entire Safe balance instead of redeemed amount

Severity: Low
Affected Contracts: `MoonwellHandler.sol`
Affected Functions: `switchFrom` (L210-220), `withdraw` (L424-431)

Summary

The `MoonwellHandler` incorrectly utilizes `IERC20.balanceOf(onBehalfOf)` to determine the amount of tokens to transfer from the user's Safe. Because Moonwell's `redeem` functions send underlying tokens to the caller (the Safe), the handler attempts to pull the *entire* balance of that asset from the Safe. In debt swap or deleverage flows, this causes pre-existing funds in the Safe to be unintentionally transferred and potentially locked as collateral in a new protocol.

Technical Details

In `switchFrom` and `withdraw`, the handler redeems collateral from Moonwell. Since Moonwell sends tokens directly to the `onBehalfOf` address (the Safe), the handler must then transfer these tokens from the Safe to the contract managing the operation (e.g., `SafeDebtManager`).

However, the implementation pulls the full balance:

```
// MoonwellHandler.sol
uint256 currentBalance = IERC20(asset).balanceOf(onBehalfOf);
bool successTransfer = ISafe(onBehalfOf).execTransactionFromModule(
    asset, 0,
    abi.encodeCall(IERC20.transfer, (address(this), currentBalance)),
    ISafe.Operation.Call
);
```

If the Safe holds any amount of the same asset prior to the transaction, those funds are swept along with the redeemed collateral. In a `switchFrom` call (part of a debt swap), these excess funds are then passed to the next handler's `switchTo` function, which typically supplies the *entire* received balance to the destination protocol:

```
// Example: AaveV3Handler.sol L119
uint256 currentBalance = IERC20(collateralAssets[i].asset).balanceOf(address(this));
aaveV3Pool.supply(collateralAssets[i].asset, currentBalance, onBehalfOf, 0);
```

Impact

Pre-existing funds in a user's Safe are unintentionally committed to a lending protocol. If the user has a borrow position, these funds become locked collateral. Withdrawing them requires additional transactions and is subject to the new protocol's health factor constraints. This violates the principle of least privilege and results in a loss of fund control for the user.

Notably, the handler already correctly uses a balance-delta pattern for ETH/WETH wrapping (L176, L198) but failed to apply it to ERC20 transfers.

Recommendation

Implement a balance-delta check to ensure only the redeemed amount is transferred from the Safe.

```
+ uint256 balanceBefore = IERC20(asset).balanceOf(onBehalfOf);

  // Perform Moonwell redeem operation

+ uint256 balanceAfter = IERC20(asset).balanceOf(onBehalfOf);
+ uint256 amountToTransfer = balanceAfter - balanceBefore;
+ require(amountToTransfer > 0, "Redeem failed");

  ISafe(onBehalfOf).execTransactionFromModule(
      asset, 0,
-     abi.encodeCall(IERC20.transfer, (address(this), currentBalance)),
+     abi.encodeCall(IERC20.transfer, (address(this), amountToTransfer)),
      ISafe.Operation.Call
  );
```

Informational

[I-1] Unsafe `IERC20.transfer` usage instead of `SafeERC20.safeTransfer`

Files: FluidSafeHandler.sol, MoonwellHandler.sol

Description: Despite importing and declaring `using SafeERC20 for IERC20`, several locations use raw `IERC20(asset).transfer()` instead of `safeTransfer()`. While these transfers are for tokens the contract holds and target a Safe wallet, some ERC20 tokens (e.g., USDT) do not return a boolean on transfer, which would cause these calls to revert.

Affected locations:

- `FluidSafeHandler.switchFrom` L93: `IERC20(fromAsset).transfer(onBehalfOf, amount)`
- `FluidSafeHandler.repay` L183: `IERC20(asset).transfer(onBehalfOf, amount)`
- `FluidSafeHandler._supplyCollateral` L226, L256: `IERC20(asset).transfer(onBehalfOf, amount)`
- `MoonwellHandler.switchTo` L245: `IERC20(collateralAssets[i].asset).transfer(onBehalfOf, currentBalance)`
- `MoonwellHandler.supply` L305: `IERC20(asset).transfer(onBehalfOf, amount)`

Recommendation: Replace all `transfer()` calls with `safeTransfer()`.

[I-2] Inconsistent error handling: mix of `require` strings and custom errors

File: MoonwellHandler.sol

Description: `MoonwellHandler` uses custom error `TokenNotRegistered()` and `MoonwellOperationFailed(uint256)` in some places, but `require` with string messages in others. This inconsistency increases gas costs (string errors are more expensive) and reduces code readability.

Recommendation: Standardize on custom errors across all handlers for gas efficiency and consistency.

[I-3] Hardcoded interest rate mode `2` (variable) in `AaveV3Handler`

File: AaveV3Handler.sol

Description: All borrow and repay calls use hardcoded interest rate mode `2` (variable rate). While Aave V3 currently defaults to variable rate, this hardcoding prevents support for stable rate borrowing in the future.

```
aaveV3Pool.borrow(asset, amount, 2, 0, onBehalfOf); // hardcoded variable
aaveV3Pool.repay(asset, amount, 2, onBehalfOf);       // hardcoded variable
```

Recommendation: Consider making the interest rate mode configurable.
