Ratehopper

SECURITY AUDIT REPORT

16 Jan 2026

Prepared by: **Shred Security**

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

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 a second round of contract implementations (PR #2) that significantly expands the protocol's position management capabilities. This update includes `deleveragePosition()` for closing leveraged positions or deleveraging by repaying debt through collateral sales, `exit()` for normal debt position exits, a wrapper contract for Safe `execTransaction()` integration, per-protocol enable/disable controls for switchFrom/To operations, and timelock protection for critical variables in the registry contract to enhance security and governance.

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 8 days in total. In this period of time, a total of 13 issues were found.

About the Project

Project Name	Ratehopper Contracts
Repository	https://github.com/RateHopper/ratehopper-contracts/pull/2/commits/f36a56084552af6daee0f1012256d032f709e8e5
Commit	f36a560
Type of Project	Intelligent DeFi platform
Lines of Code	800

Audit Timeline

Audit Start	18/12/2025
Audit End	25/12/2025
Report Published	16/01/2026

Vulnerability Summary

Severity	Count	Fixed	Acknowledged
High Risk	3	3	
Medium Risk	4	4	
Low Risk	6	3	3
Informational	0		
Total	13	10	3

Findings Summary

commit

Issue ID	Description	Severity	Status
H-1	Leveraged position creation fails if `protocolFee > 0`	High	Fixed
H-2	Multi-Collateral positions can leave assets stuck when migrating to Fluid	High	Fixed
H-3	Cross-Asset debt swaps spend fee portion, causing revert or fee loss	High	Fixed
M-1	De-whitelisted assets can still be interacted with in Aave handler	Medium	Fixed
M-2	Debt minimum threshold can be bypassed	Medium	Fixed
M-3	`amountInMax` uses wrong decimals in cross-asset debt swaps, causing under-borrow and revert	Medium	Fixed
M-4	Moonwell collateral balance upper-bound check can block exits and migrations	Medium	Fixed
L-1	Whitelisted assets without Comet addresses set for them yet can be supplied into compound regardless	Low	Acknowledged
L-2	`exit()` cannot reliably withdraw collateral after debt repayment	Low	Fixed
L-3	Timelock misalignment for emergency handler updates	Low	Acknowledged
L-4	Hardcoded minimum debt amount threshold	Low	Acknowledged
L-5	Protocol Fee Rounding Allows Fee Abuse	Low	Fixed
L-6	Use of `transferFrom()` instead of `safeTransferFrom()` for ERC20 transfers	Low	Fixed

Findings

High Severity

[H-1] Leveraged position creation fails if `protocolFee > 0`

Affected File: contracts/LeveragedPosition.sol
Severity: High
Impact: High
Likelihood: High

Description

In `_handleCreateCallback()`, the protocol fee transfer is attempted after all debt tokens have been swapped to collateral. The execution flow is:

1. Line 324: Borrows `amountInMax` (which includes `borrowAmount + protocolFeeAmount`) of debt tokens
2. Line 331-337: Swaps **ALL** `amountInMax` debt tokens to collateral via `swapByParaswap()`
3. Line 340-341: Repays flash loan with collateral
4. Line 344-345: Transfers remaining collateral to user
5. Line 348-350: Attempts to transfer `protocolFeeAmount` debt tokens to `feeBeneficiary`

The bug occurs at step 5: the contract has 0 debt tokens because all tokens were swapped in step 2. When `IERC20(decoded.debtAsset).safeTransfer(feeBeneficiary, protocolFeeAmount)` is called, it reverts with "ERC20: transfer amount exceeds balance" or similar error, causing the entire transaction to fail.

Vulnerable Code:

```
// Line 331-337: Swaps ALL amountInMax (including fee portion)
swapByParaswap(
    decoded.debtAsset,
    decoded.collateralAsset,
    amountInMax,  // @audit bug swaps everything including fee
    amountToRepay,
    decoded.paraswapParams.swapData
);

// Line 348-350: Tries to transfer fee (but contract has 0 balance)
if (protocolFee > 0 && feeBeneficiary != address(0)) {
    IERC20(decoded.debtAsset).safeTransfer(feeBeneficiary, protocolFeeAmount);  //rever
ts with 0 amount
}
```

Impact

All leveraged position creation fails when `protocolFee > 0`

Proof of Concept

- Test file: `test/protocolFeeBug.test.ts`

```
import { loadFixture } from "@nomicfoundation/hardhat-toolbox/network-helpers";
const { expect } = require("chai");
import { ethers } from "hardhat";
import { HardhatEthersSigner } from "@nomicfoundation/hardhat-ethers/signers";
import { LeveragedPosition } from "../typechain-types";
import { abi as ERC20_ABI } from "@openzeppelin/contracts/build/contracts/ERC20.json";
import { approve, fundSignerWithETH, getDecimals, getParaswapData } from "./utils";
import {
    USDC_ADDRESS,
    cbETH_ADDRESS,
    TEST_ADDRESS,
    Protocols,
    ETH_USDC_POOL,
    DEFAULT_SUPPLY_AMOUNT,
} from "./constants";
import { AaveV3Helper } from "./protocols/aaveV3";
import { deployLeveragedPositionContractFixture } from "./deployUtils";

describe("Protocol Fee Transfer Failure", function () {
    let leveragedPosition: LeveragedPosition;
    let impersonatedSigner: HardhatEthersSigner;
    let deployedContractAddress: string;
    let aaveV3Helper: AaveV3Helper;
    let feeBeneficiary: HardhatEthersSigner;
    let owner: HardhatEthersSigner;

    const defaultTargetSupplyAmount = "0.002";
    const protocolFee = 50;

    beforeEach(async function () {
        impersonatedSigner = await ethers.getImpersonatedSigner(TEST_ADDRESS);
        await fundSignerWithETH(TEST_ADDRESS, "1.0");

        const signers = await ethers.getSigners();
        owner = signers[0];
        feeBeneficiary = signers[1];

        aaveV3Helper = new AaveV3Helper(impersonatedSigner);

        const leveragedPositionContract = await loadFixture(deployLeveragedPositionCont
ractFixture);
        deployedContractAddress = await leveragedPositionContract.getAddress();
        leveragedPosition = await ethers.getContractAt("LeveragedPosition", deployedCon
tractAddress, impersonatedSigner);

        const ownerContract = leveragedPosition.connect(owner);
        await ownerContract.setProtocolFee(protocolFee);
        await ownerContract.setFeeBeneficiary(await feeBeneficiary.getAddress());
    });

    it("Should revert when protocolFee > 0 due to insufficient balance", async function
() {
        const collateralAddress = cbETH_ADDRESS;
```

```
        const debtAsset = USDC_ADDRESS;
        const principleAmount = Number(DEFAULT_SUPPLY_AMOUNT);
        const targetAmount = Number(defaultTargetSupplyAmount);

        const collateralDecimals = await getDecimals(collateralAddress);
        const debtDecimals = await getDecimals(debtAsset);

        await approve(collateralAddress, deployedContractAddress, impersonatedSigner);
        await aaveV3Helper.approveDelegation(debtAsset, deployedContractAddress);

        const parsedTargetAmount = ethers.parseUnits(targetAmount.toString(), collatera
lDecimals);
        const parsedPrincipleAmount = ethers.parseUnits(principleAmount.toString(), col
lateralDecimals);
        const diffAmount = parsedTargetAmount - parsedPrincipleAmount;

        const paraswapData = await getParaswapData(
            collateralAddress,
            debtAsset,
            deployedContractAddress,
            diffAmount
        );

        const srcAmount = paraswapData.srcAmount;
        const borrowAmount = srcAmount + 1n;
        const expectedFeeAmount = (borrowAmount * BigInt(protocolFee)) / 10000n;

        const debtToken = new ethers.Contract(debtAsset, ERC20_ABI, impersonatedSigne
r);

        await expect(
            leveragedPosition.createLeveragedPosition(
                ETH_USDC_POOL,
                Protocols.AAVE_V3,
                collateralAddress,
                parsedPrincipleAmount,
                parsedTargetAmount,
                debtAsset,
                impersonatedSigner.address,
                "0x",
                paraswapData
            )
        ).to.be.reverted;

        const contractBalance = await debtToken.balanceOf(deployedContractAddress);
        const feeBalance = await debtToken.balanceOf(await feeBeneficiary.getAddress
());

        console.log(`expectedFee: ${ethers.formatUnits(expectedFeeAmount, debtDecimal
s)}, actualFee: ${ethers.formatUnits(feeBalance, debtDecimals)}, contractBalance: ${eth
ers.formatUnits(contractBalance, debtDecimals)}`);

        expect(contractBalance).to.equal(0);
        expect(feeBalance).to.equal(0);
    });
});
```

Recommendation

Move the fee transfer to occur **before** the swap, immediately after borrowing:

[H-2] Multi-Collateral positions can leave assets stuck when migrating to Fluid

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

Description

The debt migration flow supports multi-collateral positions when switching from protocols like Compound V3, but switching to Fluid only processes the first collateral entry (`collateralAssets[0]`). Any additional collateral assets withdrawn from the source protocol are silently ignored and can remain stuck inside `SafeDebtManager` .

In `CompoundHandler.switchFrom()` , all collateral assets are withdrawn:

```
        _validateCollateralAssets(collateralAssets);
        for (uint256 i = 0; i < collateralAssets.length; i++) {
            require(registry.isWhitelisted(collateralAssets[i].asset), "Collateral asse
t is not whitelisted");
            fromComet.withdrawFrom(onBehalfOf, address(this), collateralAssets[i].asse
t, collateralAssets[i].amount);
        }
```

However, `FluidSafeHandler.switchTo()` only supplies/borrows using the first collateral asset:

```
        // use balanceOf() because collateral amount is slightly decreased when switchi
ng from Fluid
        uint256 currentBalance = IERC20(collateralAssets[0].asset).balanceOf(address(th
is));
        require(
                currentBalance < (collateralAssets[0].amount * 101) / 100,
                "Current balance is more than collateral amount + buffer"
            );

        bytes memory returnData = _supplyCollateral(vaultAddress, nftId, collateralAsse
ts[0].asset, currentBalance, onBehalfOf);
```

There is also no sweep/refund mechanism in `SafeDebtManager.uniswapV3FlashCallback()` to return unused collateral to the user after a successful migration.

Example Scenario

Lets say a user has an over-collateralized Compound position with multiple collateral assets, e.g. `[WETH, wstETH]`, and migrates to Fluid:

1. `CompoundHandler.switchFrom()` withdraws both collaterals into `SafeDebtManager`.
2. `FluidSafeHandler.switchTo()` only supplies `collateralAssets[0]`.
3. Because the position is sufficiently over-collateralized, the Fluid position is created successfully and remains healthy using only the first collateral asset.
4. The remaining collateral token(s) are neither supplied to Fluid nor returned to the user, and remain stuck in `SafeDebtManager`.

This makes the issue particularly dangerous because the migration succeeds and the resulting Fluid position appears healthy, while the user silently loses access to part of their collateral.

Impact

Users migrating multi-collateral positions to Fluid can silently lose access to one or more collateral assets, which remain stuck in `SafeDebtManager` while the migration succeeds and the new Fluid position appears healthy.

Recommendation

We suggest to revert early when migrating to Fluid if `collateralAssets.length > 1`.

```
    function switchTo(
        address toAsset,
        uint256 amount,
        address onBehalfOf,
        CollateralAsset[] memory collateralAssets,
        bytes calldata extraData
    ) public override onlyAuthorizedCaller(onBehalfOf) {
+       require(collateralAssets.length == 1, "Fluid only supports one collateral asse
t");
```

[H-3] Cross-Asset debt swaps spend fee portion, causing revert or fee loss

Affected File: contracts/SafeDebtManager.sol
Severity: High
Impact: High
Likelihood: High

Description

In the flash callback for cross-asset debt swaps, the contract computes:

- `amountTotal = amountInMax + flashloanFee + protocolFeeAmount`
- Then calls `swapByParaswap` with `amountTotal` of the destination debt asset (`toAsset`), swapping **the entire amount**, including the protocol fee portion.

Because all `toAsset` is consumed by the swap, there is no `toAsset` balance left to pay `protocolFeeAmount` to `feeBeneficiary`, causing either:

- A revert on the fee transfer (if balance is zero/insufficient), or
- Silent zero-fee collection if the transfer is skipped due to balance checks.

Relevant code (flash callback):

```
contracts/SafeDebtManager.sol
uint256 amountInMax = decoded.paraswapParams.srcAmount == 0 ? decoded.amount : decoded.
paraswapParams.srcAmount;
uint256 amountTotal = amountInMax + flashloanFee + protocolFeeAmount;
...
swapByParaswap(
    decoded.toAsset,
    decoded.fromAsset,
    amountTotal,
    amountToRepay,
    decoded.paraswapParams.swapData
);
...
if (protocolFee > 0 && feeBeneficiary != address(0)) {
    IERC20(decoded.toAsset).safeTransfer(feeBeneficiary, protocolFeeAmount);
}
```

Impact

Cross-asset debt migrations with fees enabled can fail at runtime (revert in flash callback) or result in zero protocol fees collected. This blocks migrations for users and undermines fee capture for the protocol.

Recommendation

Reserve the fee portion before swapping: keep `protocolFeeAmount` (and any buffer) un-swapped.

Medium Severity

[M-1] De-whitelisted assets can still be interacted with in Aave handler

Affected File: contracts/protocols/aaveV3Handler.sol
Severity: Medium
Impact: Medium
Likelihood: Medium

Description

While interacting with third party lending protocols such as Aave, Morpho, Compound etc, ratehopper whitelists a set of assets (USDC, WETH, cbBTC etc) that can be supplied as collateral or borrowed as debt tokens. If an asset class were to become vulnerable, no longer reliable, or has very bad liquidity related issues e.g depeg, ratehopper can halt usage of such asset class as collateral or debt token.

However, in the Aave handler (`aaveV3Handler`) there is no such check to ensure the asset being interacted with for supplying as collateral to Aave or being borrowed as debt token from Aave is whitelisted or not. Thus, assets that are no longer trusted e.g stablecoins when depegged and would be dewhitelisted in Morpho handler for example, would still be possible to be borrowed or used as collateral in the Aave handler.

```
function supply(address asset, uint256 amount, address onBehalfOf, bytes calldata /* ex
traData */) external override onlyAuthorizedCaller(onBehalfOf) {
        IERC20(asset).forceApprove(address(aaveV3Pool), amount);
        aaveV3Pool.supply(asset, amount, onBehalfOf, 0);
        IERC20(asset).forceApprove(address(aaveV3Pool), 0);
    }
```

In the `supply()` function above, there is no such: `require(registry.isWhitelisted(asset), "Asset is not whitelisted");` check present whereas the registry should be checked against supported assets when users try to interact with any loan operation such as `borrow()` , `supply()` , `repay()`

Impact

Assets that have been intentionally de-whitelisted due to risk can still be supplied or borrowed via the Aave handler, bypassing protocol risk controls and exposing users and the system to assets the registry explicitly intended to block.

Recommendation

Add the check below in the functions listed above:

```
require(registry.isWhitelisted(asset), "Asset is not whitelisted");
```

[M-2] Debt minimum threshold can be bypassed

Affected File: contracts/SafeDebtManager.sol
Severity: Medium
Impact: Medium
Likelihood: Medium

Description

`executeDebtSwap()` enforces a minimum debt threshold by requiring `_amount >= 10000`:

```
        require(_amount >= 10000, "Debt amount below minimum threshold");
```

However, when `_amount` is set to `type(uint256).max`, the function replaces `_amount` with the actual debt value returned by `getDebtAmount()`, **without re-validating the minimum threshold**:

```
        if (_amount == type(uint256).max) {
            debtAmount = IProtocolHandler(fromHandler).getDebtAmount(_fromDebtAsset, _o
nBehalfOf, _extraData[0]);
        }
```

As a result, a user can open a very small debt position (below the intended minimum) and still successfully call `executeDebtSwap()` by passing `MaxUint256`, bypassing the minimum debt restriction.

Impact

Users can bypass the intended minimum debt size by passing `type(uint256).max` allowing spammy or unsupported debt swaps that the minimum threshold was designed to prevent.

Recommendation

After fetching the actual debt amount when `_amount == type(uint256).max`, enforce the same minimum threshold check on the resolved debtAmount:

```
        if (_amount == type(uint256).max) {
            debtAmount = IProtocolHandler(fromHandler).getDebtAmount(_fromDebtAsset, _o
nBehalfOf, _extraData[0]);
+           require(debtAmount >= 10000, "Debt amount below minimum threshold");
        }
```

[M-3] `amountInMax` uses wrong decimals in cross-asset debt swaps, causing under-borrow and revert

Affected File: contracts/SafeDebtManager.sol
Severity: Medium
Impact: Medium
Likelihood: Medium

Description

During cross-asset debt swaps (`fromAsset != toAsset`), the callback converts the flashloan fee (and protocol fee) into `toAsset` decimals:

```
        } else {
            // Convert amount to toAsset decimals first
            uint256 amountInToAssetDecimals = decoded.amount;
            if (decimalDifference > 0) {
                amountInToAssetDecimals = decoded.amount / (10 ** uint256(decimalDiffer
ence));
            } else if (decimalDifference < 0) {
                amountInToAssetDecimals = decoded.amount * (10 ** uint256(-decimalDiffe
rence));
            }
            protocolFeeAmount = (amountInToAssetDecimals * protocolFee) / 10000;
        }
```

However, `amountInMax` is set in `fromAsset` decimals when `paraswapParams.srcAmount == 0` :

```
        uint256 amountInMax = decoded.paraswapParams.srcAmount == 0 ? decoded.amount :
 decoded.paraswapParams.srcAmount;
        uint256 amountTotal = amountInMax + flashloanFee + protocolFeeAmount;
```

This mixes units:

- `amountInMax` is `fromAsset` decimals

- `flashloanFee + protocolFeeAmount` are `toAsset` decimals

As a result, `amountTotal` is incorrect (typically far too small when swapping from lower-decimal assets like USDC(6) to higher-decimal assets like DAI(18)), so the destination protocol under-borrows. The subsequent Paraswap conversion then fails to produce enough `fromAsset` to repay the flashloan, reverting the swap.

Example: USDC(6) → DAI(18)
`decoded.amount = 1100e6` but should contribute ~ `1100e18` to the `toAsset`-denominated `amountTotal`.

Impact

Cross-asset debt migrations can under-borrow on the destination protocol due to mixed decimal units, causing flashloan repayment failures and transaction reverts for otherwise valid swaps.

Recommendation

We should scale up `amountInMax` by the decimal difference if switching from debt asset of lesser decimals to debt asset of higher decimals. We should also do the vice-versa when switching from debt asset of higher decimals to debt asset of lower decimals by scaling `amountInMax` down by decimal difference.

[M-4] Moonwell collateral balance upper-bound check can block exits and migrations

Affected File: contracts/protocolsSafe/MoonwellHandler.sol, contracts/protocols/morphoHandler.sol, contracts/protocols/aaveV3Handler.sol, contracts/protocols/compoundHandler.sol, contracts/protocolsSafe/FluidSafeHandler.sol

Severity: Medium
Impact: Medium
Likelihood: High

Description

The Moonwell handler enforces a strict upper-bound check on collateral balances using a fixed ~1% buffer:

```
            require(
                currentBalance * 100 < collateralAssets[i].amount * 101,
                "Current balance is more than collateral amount + buffer"
            );
```

This check is applied in both `switchFrom` and `switchTo`, but critically, it operates on total token balances, not on the amount actually redeemed during the operation.

Handler-balance griefing / donation attack

In `switchTo`, the check is performed against the contract's balance:

```
            // use balanceOf() because collateral amount is slightly decreased when swi
tching from Fluid
            uint256 currentBalance = IERC20(collateralAssets[i].asset).balanceOf(addres
s(this));
            require(
                currentBalance * 100 < collateralAssets[i].amount * 101,
                "Current balance is more than collateral amount + buffer"
            );
```

Because this uses the entire `SafeDebtManager` balance, a third party can grief the system by donating extra tokens to the `SafeDebtManager` contract, or leftover balances from prior migrations can accumulate. Even a small donation or residual balance is sufficient to push `currentBalance` above the 1% buffer threshold.

Once this happens:

- `switchTo` reverts
- legitimate debt migrations into Moonwell fail

- no privileged access is required by the attacker

Safe-balance drift blocking exits

A similar check exists in `switchFrom`, but applied to the Safe's balance:

```
            uint256 currentBalance = IERC20(collateralAssets[i].asset).balanceOf(onBeha
lfOf);
            require(
                currentBalance * 100 < collateralAssets[i].amount * 101,
                "Current balance is more than collateral amount + buffer"
            );
```

This check incorrectly assumes the Safe holds only the collateral redeemed during this operation. In practice, balances can exceed expectations due to:

- interest accrual
- reward distributions
- rebasing tokens
- leftovers from prior migrations
- unrelated user transfers

When this happens, exiting or migrating a Moonwell position becomes impossible, even though the position itself is healthy.

Impact

- An attacker can grief the system by transferring tokens to the `SafeDebtManager` contract, breaking `switchTo` for affected assets and blocking migrations for all users until the admin manually withdraws the tokens from the contract.
- Users with grown or pre-existing collateral balances cannot exit Moonwell or migrate positions, leaving funds effectively stuck through the intended workflow.

Recommendation

Replace the upper-bound revert with logic that is resilient to balance drift and external transfers:

- Measure delta balances (before vs. after redemption) instead of total balances, or
- Remove the upper-bound check entirely and retain only a lower-bound sanity check

---

Low Severity

[L-1] Whitelisted assets without Comet addresses set for them yet can be supplied into compound regardless

Affected File: contracts/protocols/compoundHandler.sol
Severity: Low
Impact: Low
Likelihood: Low

Description

Compound V3 introduces Comet thus, in Compound, Comet refers to the V3 compound money markets. The ratehopper contract specifically the `compoundHandler` handles entering/exit into compound lending markets. Also, assets that can be deposited as collateral or borrowed are whitelisted in the registry contract (ProtocolRegistry) where all handlers reference to check for whitelisted assets, assets whose comet addresses have been set or not (in the case of compound).

The issue is that one such asset class e.g WETH being whitelisted in the ProtocolRegistry contract doesn't necessarily mean we can supply it into compound markets yet. Because for the case of compound, the comet address needs to be set for this asset class first. However, in `supply()` and `withdraw()` of the compoundHandler, we allow arbitrary comet addresses supplied by the user without validating the supplied comet address against what we have set for that asset class in `registry.getCContract(token)`.

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

@>        address cContract = abi.decode(extraData, (address)); // @audit arbitrary com
et address supplied by user not verified against the comet setup for `asset` in `getCCo
ntract(asset);`
        require(cContract != address(0), "Invalid comet address");

        IERC20(asset).forceApprove(address(cContract), amount);
        // supply collateral
        IComet(cContract).supplyTo(onBehalfOf, asset, amount);
        IERC20(asset).forceApprove(address(cContract), 0);
    }

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

```
        // Decode the comet address from extraData
@>        address cContract = abi.decode(extraData, (address)); // @audit arbitrary com
et address supplied by user not verified against the comet setup for `asset` in `getCCo
ntract(asset);`
        require(cContract != address(0), "Invalid comet address");

        // Withdraw collateral from user's position to this contract
        IComet comet = IComet(cContract);
        comet.withdrawFrom(onBehalfOf, address(this), asset, amount);
    }
```

Consider the scenario whereby WETH is a whitelisted asset class:

- In Aave, and Morpho we can supply WETH as collateral into those protocols and perhaps borrow USDC
- For Compound, there is no comet address setup for WETH yet so technically we are disallowing interaction with any compound v3 market at this time (e.g due to a flaw in compound or anything else)
- Regardless of this, a user can just grab the comet address and encode it into the extraData argument provided.
- Now, regardless of ratehopper not setting up the comet address for WETH & USDC yet, the user will be able to supply WETH collateral into Compound V3 USDC-WETH market

Recommendation

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

        address cContract = abi.decode(extraData, (address));
        require(cContract != address(0), "Invalid comet address");

+       address wcContract = getCContract(asset);
+       require(cContract == wcContract, "Invalid comet address");

        IERC20(asset).forceApprove(address(cContract), amount);
        // supply collateral
        IComet(cContract).supplyTo(onBehalfOf, asset, amount);
        IERC20(asset).forceApprove(address(cContract), 0);
    }
```

Replicate the above diff in the `withdraw()` function as well.

[L-2] `exit()` cannot reliably withdraw collateral after debt repayment

Affected File: contracts/SafeDebtManager.sol
Severity: Low
Impact: Low
Likelihood: Low

Description

The `exit()` function is designed to both repay debt and optionally withdraw collateral. Users may choose to repay their debt without withdrawing collateral by setting `_withdrawCollateral = false`. However, the contract does not provide a reliable way to withdraw collateral later using the same function.

The function enforces a minimum `_debtAmount`:

```
        require(_debtAmount >= 10000, "Debt amount below minimum threshold");
```

This prevents users from later calling `exit()` with `_withdrawCollateral = true` and `_debtAmount = 0` to perform a withdrawal-only operation.

A workaround is to call `exit()` with `_debtAmount = type(uint256).max`, which resolves to `repayAmount = 0` once the debt has already been fully repaid.

```
        if (_debtAmount == type(uint256).max) {
            uint256 debtAmount = IProtocolHandler(handler).getDebtAmount(_debtAsset, _o
nBehalfOf, _extraData);
```

This approach works on protocols such as Morpho and Compound, where repaying zero does not revert.

However, this pattern fails on Aave V3, which explicitly reverts on zero-amount repayments.

Aave's internal repayment validation enforces a non-zero repay amount:

```
  function validateRepay(
    DataTypes.ReserveCache memory reserveCache,
    uint256 amountSent,
    DataTypes.InterestRateMode interestRateMode,
    address onBehalfOf,
    uint256 stableDebt,
    uint256 variableDebt
  ) internal view {
    require(amountSent != 0, Errors.INVALID_AMOUNT);
```

As a result, users who previously repaid their debt without withdrawing collateral cannot later withdraw collateral through `exit()` when the destination protocol is Aave V3.

Recommendation

Decouple debt repayment from collateral withdrawal logic.

Possible fixes include:

- Allowing `_debtAmount == 0` when `_withdrawCollateral == true`, and skipping the repay path entirely
- Adding a dedicated collateral-withdrawal function that does not require a debt amount

---

[L-3] Timelock misalignment for emergency handler updates

Affected File: contracts/LeveragedPosition.sol
Severity: Low
Impact: Low
Likelihood: Low

Description

The `setProtocolHandler` function which is implemented inside `SafeDebtManager.sol` and `LeveragedPosition.sol`, is documented as allowing handler updates when a bug is found:

```
    /// @notice Updates the handler address for a specific protocol
    /// @param _protocol The protocol to update the handler for
    /// @param _handler The new handler address
    /// @dev Only callable by addresses with CRITICAL_ROLE in ProtocolRegistry. For pro
duction, should be behind a timelock.
    /// @dev Allows updating handlers if a bug is found or handler needs upgrade.
    function setProtocolHandler(Protocol _protocol, address _handler) external onlyCrit
icalRole {
        require(_handler != address(0), "Invalid handler address");
        address oldHandler = protocolHandlers[_protocol];
        protocolHandlers[_protocol] = _handler;
        emit ProtocolHandlerUpdated(_protocol, oldHandler, _handler);
    }
```

but it is intended to be protected by a timelock in production. Timelocks introduce execution delays, which conflicts with the stated purpose of enabling rapid response to critical bugs. This creates an inconsistency between the function’s documented intent and its recommended governance configuration.

Recommendation

It is recommended not to place this function behind a timelock if it is intended to be used for immediate bug fixes.

[L-4] Hard coded minimum debt amount threshold

Affected File: contracts/SafeDebtManager.sol
Severity: Low
Impact: Low

Description

The contract enforces a hard-coded minimum debt amount (`_amount >= 10000`) when executing debt-related operations. This value is expressed in raw token units, making it dependent on token decimals and asset type. As a result, the check may be ineffective for high-decimal tokens or overly restrictive for low-decimal tokens.

- For a 6-decimal token (USDC) → 10000 = 0.01 USDC
- For an 18-decimal token (DAI) → 10000 = 1e-14 DAI

Recommendation

Avoid using hard-coded, raw-unit thresholds for economic constraints. It is recommended to adopt one of the following approaches:

- Configurable per-asset minimum

```
    mapping(address => uint256) public minDebtAmount;

    require(_amount >= minDebtAmount[_fromDebtAsset], "Below minimum debt amount");
```

- Value-based minimum using normalization

```
    uint256 normalizedAmount = _amount * 1e18 / (10 ** IERC20Metadata(_fromDebtAsset).d
ecimals());
    require(normalizedAmount >= MIN_DEBT_VALUE_18, "Below minimum debt value");
```

[L-5] Protocol Fee Rounding Allows Fee Abuse

Affected File: contracts/SafeDebtManager.sol
Severity: Low
Impact: Low
Likelihood: Low

Description

Fee calculations use floor division when converting between different decimal precisions. Small amounts result in zero fees, allowing users to avoid protocol fees by splitting large operations into many small transactions.

In `SafeDebtManager`, fees are calculated after decimal conversion:

```
amountInToAssetDecimals = decoded.amount / (10 ** decimalDifference);  // Floors to 0 f
or small amounts
protocolFeeAmount = (amountInToAssetDecimals * protocolFee) / 10000;  // Results in 0
```

Example: With `amount=10` (1 decimal) converting to 18 decimals and `fee=20` (0.2%):

- `10 / 10^17 = 0` (floor division)
- `(0 * 20) / 10000 = 0`
- Fee charged: 0

Users can split a large withdrawal into 100+ small transactions, each paying zero fees instead of the intended fee on the total amount.

Recommendation

Use ceiling division for fee calculations or enforce a minimum fee:

```
// Calculate fee before decimal conversion
uint256 feeInFromAsset = (decoded.amount * protocolFee) / 10000;
// Then convert the fee amount, not the base amount

// Or use ceiling division
uint256 numerator = amountInToAssetDecimals * protocolFee;
protocolFeeAmount = (numerator + 9999) / 10000; // Ceiling
```

Alternatively, enforce minimum fee:

```
uint256 feeAmount = (amount * fee) / 10000;
if (feeAmount == 0 && amount > 0 && fee > 0) {
```

```
        feeAmount = 1; // Minimum 1 wei
    }
```

[L-6] Use of `transferFrom()` instead of `safeTransferFrom()` for ERC20 transfers

Affected File: contracts/LeveragedPosition.sol
Severity: Low
Impact: Low
Likelihood: Low

Description

There are token transfers in the contract that rely on the raw `transfer()` / `transferFrom()` call instead of `safeTransfer()` / `safeTransferFrom()` .

For example, in `createLeveragedPosition()` :

```
        IERC20(_collateralAsset).transferFrom(_onBehalfOf, address(this), _principleCol
    lateralAmount);
```

Non-standard ERC20 tokens may return false on failure instead of reverting. When using `transferFrom()` directly, such failures may go unnoticed, potentially causing the function to continue execution under the assumption that the transfer succeeded, leading to unexpected behavior or downstream reverts.

Recommendation

Use `safeTransferFrom()` from OpenZeppelin’s SafeERC20 library to ensure that token transfers either succeed or revert explicitly.
