comet (Compound Finance)

Benchmark Security Review Performed by Bug Hunter

16.05.2025

Table of Contents

About Bug Hunter

Bug Hunter is an automated code-review tool for Solidity smart contracts, developed by Truscova GmbH. It uses proprietary algorithms and advanced code-analysis techniques to identify potential vulnerabilities efficiently.

Bug Hunter is designed to support developers in the Ethereum ecosystem by delivering consistent, scalable assessments that align with best practices in Web 3.0 security. Reports are generated via the Bug Hunter tool and are accessible to clients through a secure dashboard web interface or relevant Git repositories.

Bug Hunter is a product of Truscova, a security R&D company headquartered in Bremen, Germany. Founded in late 2022, Truscova specializes in formal verification, fuzzing, dynamic and static analysis, and security tooling. Its founding team includes researchers with 35+ books, several patents, and over 20 000 citations in software testing, verification, security, and machine learning.

To explore more about Bug Hunter, visit:

- Bug Hunter Website
- Bug Hunter Documentation
- Example Reports

To explore more about Truscova, visit:

- Truscova Website
- X
- LinkedIn

Notices and Remarks

Copyright and Distribution

This security-review report was generated with Truscova's Bug Hunter tool to benchmark its performance on selected Yield Farming projects. It is made available through the Bug Hunter dashboard and relevant Git repositories. Users may choose to share or publish this report at their own discretion.

Test Coverage Disclaimer

This security review was conducted exclusively by Bug Hunter, an automated code-review tool for Solidity codebases. Bug Hunter uses a combination of proprietary algorithms and programmatic techniques to identify potential vulnerabilities in Solidity smart contracts.

The findings in this report should not be considered a complete or exhaustive list of all possible security issues in the reviewed codebase and are dependent on Bug Hunter's coverage of detector rules. The full list of rules covered by Bug Hunter automated code review is available at https://docs.bughunter.live/.

Executive Summary

Overview and Scope

A codebase from project comet was reviewed for security vulnerabilities by Bug Hunter with the following details:

- GitHub Repository: https://github.com/compound-finance/comet
  Commit hash: `71aec025e1fcf9c16a1aeb20a85eddae5a5d5581`

Bug Hunter reviewed the following files:

Comet.sol CometExt.sol CometCore.sol

Summary of Findings

The uncovered vulnerabilities identified during the security review are summarised in the table below:

Severity	Count
High	0
Medium	3
Low	3

Project Summary

Contact Information

For support, contact us at support@bughunter.live

Project Timeline

Date	Event
16.05.2025	Project submitted
16.05.2025	Security review concluded

Detailed Findings

1. Incomplete Sanitization of Configuration Parameters in Constructor

Severity	Low
Finding ID	BH-comet-01
Target	contracts/Comet.sol
Function name	constructor

Description

The constructor of the contract performs some sanity checks on the `Configuration` struct but fails to comprehensively validate all fields. Critical parameters such as interest rate slopes, reserve rate, tracking speeds, and asset configurations are not fully validated, leaving room for misconfiguration or malicious setup. For example, invalid or extreme interest rate values could result in dysfunctional lending behavior, and poorly configured asset settings may compromise protocol safety.

```
constructor(Configuration memory config) {
    // Sanity checks
    uint8 decimals_ = ERC20(config.baseToken).decimals();
    if (decimals_ > MAX_BASE_DECIMALS) revert BadDecimals();
    if (config.storeFrontPriceFactor > FACTOR_SCALE) revert BadDiscount();
    if (config.assetConfigs.length > MAX_ASSETS) revert TooManyAssets();
    if (config.baseMinForRewards == 0) revert BadMinimum();
    if (AggregatorV3Interface(config.baseTokenPriceFeed).decimals() !=
PRICE_FEED_DECIMALS) revert BadDecimals();
    // XXX other sanity checks? for rewards?

    // Copy configuration
    unchecked {
        governor = config.governor;
        pauseGuardian = config.pauseGuardian;
        baseToken = config.baseToken;
        baseTokenPriceFeed = config.baseTokenPriceFeed;
        extensionDelegate = config.extensionDelegate;
        storeFrontPriceFactor = config.storeFrontPriceFactor;

        decimals = decimals_;
```

```
        baseScale = uint64(10 ** decimals_);
        trackingIndexScale = config.trackingIndexScale;
        accrualDescaleFactor = baseScale / 1e6;

        baseMinForRewards = config.baseMinForRewards;
        baseTrackingSupplySpeed = config.baseTrackingSupplySpeed;
        baseTrackingBorrowSpeed = config.baseTrackingBorrowSpeed;

        baseBorrowMin = config.baseBorrowMin;
        targetReserves = config.targetReserves;
    }

    // Set interest rate model configs
    unchecked {
        kink = config.kink;
        perSecondInterestRateSlopeLow = config.perYearInterestRateSlopeLow /
SECONDS_PER_YEAR;
        perSecondInterestRateSlopeHigh = config.perYearInterestRateSlopeHigh /
SECONDS_PER_YEAR;
        perSecondInterestRateBase = config.perYearInterestRateBase /
SECONDS_PER_YEAR;
        reserveRate = config.reserveRate;
    }

    // Set asset info
    numAssets = uint8(config.assetConfigs.length);

    (asset00_a, asset00_b) = _getPackedAsset(config.assetConfigs, 0);
    (asset01_a, asset01_b) = _getPackedAsset(config.assetConfigs, 1);
    (asset02_a, asset02_b) = _getPackedAsset(config.assetConfigs, 2);
    (asset03_a, asset03_b) = _getPackedAsset(config.assetConfigs, 3);
    (asset04_a, asset04_b) = _getPackedAsset(config.assetConfigs, 4);
    (asset05_a, asset05_b) = _getPackedAsset(config.assetConfigs, 5);
    (asset06_a, asset06_b) = _getPackedAsset(config.assetConfigs, 6);
    (asset07_a, asset07_b) = _getPackedAsset(config.assetConfigs, 7);
    (asset08_a, asset08_b) = _getPackedAsset(config.assetConfigs, 8);
    (asset09_a, asset09_b) = _getPackedAsset(config.assetConfigs, 9);
    (asset10_a, asset10_b) = _getPackedAsset(config.assetConfigs, 10);
    (asset11_a, asset11_b) = _getPackedAsset(config.assetConfigs, 11);
    (asset12_a, asset12_b) = _getPackedAsset(config.assetConfigs, 12);
    (asset13_a, asset13_b) = _getPackedAsset(config.assetConfigs, 13);
    (asset14_a, asset14_b) = _getPackedAsset(config.assetConfigs, 14);

    // Initialize storage
```

```
    initializeStorage();
}
```

Recommendation

Implement additional sanitization checks to validate all critical parameters in the `Configuration` struct. This includes bounds checking on interest rate parameters, reserve rate (e.g., must be ≤ 1e18), asset-specific values (e.g., non-zero asset addresses, valid price feeds, etc.), and ensuring consistency between related configuration values. This will improve protocol robustness and prevent silent misconfiguration.

2. Unchecked ecrecover Return Value May Yield Zero Address

Severity	Medium
Finding ID	BH-comet-02
Target	contracts/CometExt.sol
Function name	allowBySig

Description

The `ecrecover` function can return the zero address (address(0)) if the signature is invalid or malformatted. The current implementation does not explicitly check whether the returned address is the zero address, which can result in a false match with the provided `owner` address in specific edge cases, particularly if the `owner` address is zero or poorly validated elsewhere in the code.

```
function allowBySig(
    address owner,
    address manager,
    bool isAllowed_,
    uint256 nonce,
    uint256 expiry,
    uint8 v,
    bytes32 r,
    bytes32 s
) external {
    if (uint256(s) > MAX_VALID_ECDSA_S) revert InvalidValueS();
    // v ∈ {27, 28} (source: https://ethereum.github.io/yellowpaper/paper.pdf #308)
    if (v != 27 && v != 28) revert InvalidValueV();

    bytes32 domainSeparator = keccak256(
        abi.encode(
            DOMAIN_TYPEHASH,
            keccak256(bytes(name)),
            keccak256(bytes(version)),
            block.chainid,
            address(this)
        )
    );
```

```
    bytes32 structHash = keccak256(
        abi.encode(
            AUTHORIZATION_TYPEHASH,
            owner,
            manager,
            isAllowed_,
            nonce,
            expiry
        )
    );

    bytes32 digest = keccak256(
        abi.encodePacked("\x19\x01", domainSeparator, structHash)
    );

    address signatory = ecrecover(digest, v, r, s);
    if (signatory == address(0)) revert InvalidSignatory();
    if (owner != signatory) revert BadSignatory();
    if (nonce != userNonce[signatory]++) revert BadNonce();
    if (block.timestamp >= expiry) revert SignatureExpired();

    allowInternal(signatory, manager, isAllowed_);
}
```

Recommendation

Add an explicit check to ensure that the recovered `signatory` is not the zero address before proceeding with comparisons or state changes. This helps prevent unintended behavior in case of signature failure.

3. Potential Integer Overflow in Supply Value Calculation

Severity	Low
Finding ID	BH-comet-03
Target	comet/contracts/CometCore.sol
Function name	presentValueSupply

Description

The function `presentValueSupply` performs a multiplication of two potentially large integers (`principalValue_` and `baseSupplyIndex_`) before division. Since both operands are cast to `uint`, this multiplication can overflow silently, especially in the absence of overflow checks in `pure` or `unchecked` context. This can result in incorrect computation of present value, potentially leading to accounting discrepancies or exploitable financial imbalances.

```
function presentValueSupply(uint64 baseSupplyIndex_, uint104 principalValue_)
internal pure returns (uint104) {
    // Use Solidity 0.8.x overflow checks (no need for SafeMath)
    uint256 result = uint256(principalValue_) * uint256(baseSupplyIndex_) /
BASE_INDEX_SCALE;
    require(result <= type(uint104).max, "Overflow in presentValueSupply");
    return uint104(result);
}
```

Recommendation

Use `SafeMath`-like overflow checks or reorder operations to reduce overflow risk (e.g., dividing before multiplying when safe). Alternatively, use Solidity 0.8.x's built-in overflow protection by avoiding `unchecked` contexts unless absolutely necessary, and ensure operands are within safe bounds before the operation.

4. Possible Overflow in Index Accrual Calculations

Severity	Low
Finding ID	BH-comet-04
Target	contracts/Comet.sol
Function name	accrueInternal

Description

The `accrueInternal` function performs index updates by multiplying rates with `timeElapsed` and applying the result to base indices (`baseSupplyIndex`, `baseBorrowIndex`, `trackingSupplyIndex`, `trackingBorrowIndex`). These operations can overflow if the computed values exceed the maximum limits of their types, especially given potentially large elapsed time or high rates. Although the function uses `safe64` for some assignments, the intermediate calculations before this casting are not protected, and overflows can occur during `mulFactor` or multiplication operations, leading to incorrect state updates.

```
function accrueInternal() internal {
    uint40 now_ = getNowInternal();
    uint256 timeElapsed = now_ - lastAccrualTime;
    if (timeElapsed > 0) {
        uint256 supplyRate = getSupplyRate();
        uint256 borrowRate = getBorrowRate();

        uint256 supplyAccrual = supplyRate * timeElapsed;
        uint256 borrowAccrual = borrowRate * timeElapsed;

        baseSupplyIndex += safe64(mulFactor(baseSupplyIndex, supplyAccrual));
        baseBorrowIndex += safe64(mulFactor(baseBorrowIndex, borrowAccrual));

        if (totalSupplyBase >= baseMinForRewards) {
            uint256 supplySpeed = baseTrackingSupplySpeed;
            uint256 supplyTrackingAccrual = supplySpeed * timeElapsed;
            trackingSupplyIndex += safe64(divBaseWei(supplyTrackingAccrual,
totalSupplyBase));
        }

        if (totalBorrowBase >= baseMinForRewards) {
            uint256 borrowSpeed = baseTrackingBorrowSpeed;
```

```
            uint256 borrowTrackingAccrual = borrowSpeed * timeElapsed;
            trackingBorrowIndex += safe64(divBaseWei(borrowTrackingAccrual,
totalBorrowBase));
        }

        lastAccrualTime = now_;
    }
}
```

Recommendation

Add overflow checks before performing multiplications and additions, particularly for `rate * timeElapsed` and `mulFactor` results. Ensure the intermediate calculations are bounded within 64-bit or 256-bit limits as required. Use Solidity 0.8.x's built-in overflow protection where possible, or validate input bounds to mitigate risk.

5. Non-Standard ERC20 `approve` Behavior May Break Integrations

Severity	Medium
Finding ID	BH-comet-05
Target	contracts/CometExt.sol
Function name	approve

Description

The `approve` function in `CometExt.sol` only allows two specific `amount` values: `0` (to revoke approval) and `type(uint256).max` (to grant unlimited approval). Any other value causes a revert. This deviates from the standard ERC20 behavior, where arbitrary allowance amounts are expected. As a result, integrations with wallets, DeFi protocols, or tools that assume standard ERC20 semantics may fail or behave unpredictably.

```
function approve(address spender, uint256 amount) external returns (bool) {
    if (amount == type(uint256).max) {
        allowInternal(msg.sender, spender, true);
    } else if (amount == 0) {
        allowInternal(msg.sender, spender, false);
    } else {
        revert BadAmount();
    }
    emit Approval(msg.sender, spender, amount);
    return true;
}
```

Recommendation

If full ERC20 compatibility is required, consider supporting arbitrary approval amounts. If restricting to `0` and `max` is a deliberate design decision, document the deviation clearly and ensure frontends and integrators are aware of this limitation to avoid integration failures.

6. baseBorrowMin Not Enforced for Transfer Destination

Severity	Medium
Finding ID	BH-comet-06
Target	contracts/Comet.sol
Function name	transferBase

Description

While `baseBorrowMin` is enforced when the sender's balance becomes negative, the receiver (`dst`) is not subject to the same check. This allows a user to indirectly create a borrow position below the minimum borrow threshold via a transfer.

```
function transferBase(address src, address dst, uint104 amount) internal {
    accrueInternal();

    uint104 totalSupplyBalance = presentValueSupply(baseSupplyIndex,
totalSupplyBase);
    uint104 totalBorrowBalance = presentValueBorrow(baseBorrowIndex,
totalBorrowBase);

    UserBasic memory srcUser = userBasic[src];
    UserBasic memory dstUser = userBasic[dst];
    int104 srcBalance = presentValue(srcUser.principal);
    int104 dstBalance = presentValue(dstUser.principal);

    (uint104 withdrawAmount, uint104 borrowAmount) =
withdrawAndBorrowAmount(srcBalance, amount);
    (uint104 repayAmount, uint104 supplyAmount) = repayAndSupplyAmount(dstBalance,
amount);

    // Note: Instead of `totalSupplyBalance += supplyAmount - withdrawAmount` to
avoid underflow errors.
    totalSupplyBalance = totalSupplyBalance + supplyAmount - withdrawAmount;
    totalBorrowBalance = totalBorrowBalance + borrowAmount - repayAmount;

    srcBalance -= signed104(amount);
    dstBalance += signed104(amount);
```

```
    totalSupplyBase = principalValueSupply(baseSupplyIndex, totalSupplyBalance);
    totalBorrowBase = principalValueBorrow(baseBorrowIndex, totalBorrowBalance);

    updateBaseBalance(src, srcUser, principalValue(srcBalance));
    updateBaseBalance(dst, dstUser, principalValue(dstBalance));

    if (srcBalance < 0) {
        if (uint104(-srcBalance) < baseBorrowMin) revert BorrowTooSmall();
        if (!isBorrowCollateralized(src)) revert NotCollateralized();
    }

    if (dstBalance < 0) {
        if (uint104(-dstBalance) < baseBorrowMin) revert BorrowTooSmall();
        if (!isBorrowCollateralized(dst)) revert NotCollateralized();
    }

    emit Transfer(src, dst, amount);
}
```

Recommendation

Add a similar check for `dstBalance < 0`, and ensure the borrow amount meets `baseBorrowMin`. Also verify that the destination account is properly collateralized.