SMART CONTRACT AUDIT REPORT

Smart Fun Smart Contract

JULY 2025

Contents

1. EXECUTIVE SUMMARY

ExVul Web3 Security was engaged by **Smart Fun** to review smart contract implementation. The assessment was conducted in accordance with our systematic approach to evaluate potential security issues based upon customer requirement. The report provides detailed recommendations to resolve the issue and provide additional suggestions or recommendations for improvement.

The outcome of the assessment outlined in chapter 3 provides the system's owners a full description of the vulnerabilities identified, the associated risk rating for each vulnerability, and detailed recommendations that will resolve the underlying technical issue.

1.1 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10] which is the gold standard in risk assessment using the following risk models:

- **Likelihood**: represents how likely a particular vulnerability is to be uncovered and exploited in the wild.
- **Impact**: measures the technical loss and business damage of a successful attack.
- **Severity**: determine the overall criticality of the risk.

Likelihood can be: High, Medium and Low and impact are categorized into: High, Medium, Low, Informational. Severity is determined by likelihood and impact and can be classified into five categories accordingly: Critical, High, Medium, Low, Informational shown in table 1.1.

Likelihood \ IMPACT	Informational	Low	Medium	High
High	INFO	MEDIUM	HIGH	CRITICAL
Medium	INFO	LOW	MEDIUM	HIGH
Low	INFO	LOW	LOW	MEDIUM

Table 1.1 Overall Risk Severity

To evaluate the risk, we will be going through a list of items, and each would be labelled with a severity category. The audit was performed with a systematic approach guided by a comprehensive assessment list carefully designed to identify known and impactful security issues. If our tool or analysis does not identify any issue, the contract can be considered safe regarding the assessed item. For any discovered issue, we might further deploy contracts on our private test environment and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.2.

- **Basic Coding Bugs**: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- **Code and business security testing**: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- **Additional Recommendations**: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Category	Assessment Item
Basic Coding Assessment	• Apply Verification Control • Authorization Access Control • Forged Transfer Vulnerability • Forged Transfer Notification • Numeric Overflow • Transaction Rollback Attack • Transaction Block Stuffing Attack • Soft Fail Attack • Hard Fail Attack • Abnormal Memo • Abnormal Resource Consumption • Secure Random Number

Advanced Source Code Scrutiny	• Asset Security • Cryptography Security • Business Logic Review • Source Code Functional Verification • Account Authorization Control • Sensitive Information Disclosure • Circuit Breaker • Blacklist Control • System API Call Analysis • Contract Deployment Consistency Check • Abnormal Resource Consumption
Additional Recommendations	• Semantic Consistency Checks • Following Other Best Practices

Table 1.2: The Full List of Assessment Items

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [14], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development.

2. FINDINGS OVERVIEW

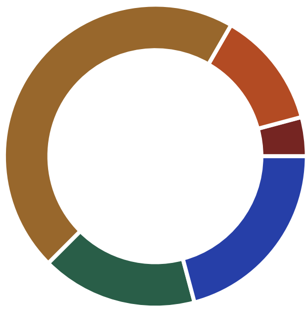
2.1 Project Info And Contract Address

Project Name	Audit Time	Language
Smart Fun	02/07/2025 - 15/07/2025	Solidity

Repository	Commit Hash
https://github.com/ora-io/smart-fun-contract/	c8aebde5c9bbddb79167f17e302ab959c4bcba01

2.2 Summary

Severity	Found
CRITICAL	1
HIGH	3
MEDIUM	11
LOW	4
INFO	5



2.3 Key Findings

Severity	Findings Title	Status
CRITICAL	Missing deadline parameter in Uniswap V3 swap calls causes complete functionality failure	Fixed
HIGH	Inconsistent Fee Calculation in executeBuyExactOut Function	Fixed
HIGH	CometFactory lacks access control allowing complete system DOS attack	Fixed
HIGH	marginSell and closeMarginSell lack pause mechanism checks	Fixed
MEDIUM	Permanently locks ETH through unrestricted receive()	Fixed
MEDIUM	Hard-coded fee tier in UniswapV3 swap operations	Fixed
MEDIUM	Unchecked ETH acceptance in non-ETH token operations	Fixed
MEDIUM	ERC20::approve will revert to some non-standard tokens	Fixed
MEDIUM	Token approval with type(uint256).max causes incompatibility	Fixed
MEDIUM	swapTokenForOraToken lacks msg.value validation	Fixed
MEDIUM	swapTokenForOraToken lacks token transfer handling	Fixed
MEDIUM	Market cap calculation parameter inconsistency	Fixed
MEDIUM	TokenConfig lacks validation allowing duplicate tokens	Fixed
MEDIUM	CometRouter Lack of Liquidator Authorization Check	Fixed
MEDIUM	Transfer Restriction Bypass via transferFrom	Fixed
LOW	Unrestricted Fee Percentage Setting	Acknowledge
LOW	LPFeeDistributor lacks slippage protection	Acknowledge

Severity	Findings Title	Status
LOW	Asset count limit inconsistency	Acknowledge
LOW	Swap router validation missing in margin trading	Acknowledge
INFO	Variable naming is inconsistent	Fixed
INFO	Migration state not reset after successful migration	Fixed
INFO	ETH address check is non-standard	Fixed
INFO	LP Fee Calculation Bypasses Uniswap V3's Accurate Values	Fixed
INFO	Price oracle precision mismatch	Fixed

Table 2.3: Key Audit Findings

3. DETAILED DESCRIPTION OF FINDINGS

3.1 Missing deadline parameter in Uniswap V3 swap calls causes complete functionality failure

SEVERITY: CRITICAL **STATUS:** Fixed

PATH:

`contracts/interfaces/external/ISwapRouter.sol`

DESCRIPTION:

The protocol's ISwapRouter interface definition lacks the deadline parameter required by the actual Uniswap V3 Router, resulting in all swap operations failing due to deadline expiration.

```
// contracts/interfaces/external/ISwapRouter.sol
struct ExactInputSingleParams {
    address tokenIn;
    address tokenOut;
    uint24 fee;
    address recipient;
    uint256 amountIn;
    uint256 amountOutMinimum;
    uint160 sqrtPriceLimitX96;
    // Missing: uint256 deadline;
}
```

Root Cause: Interface mismatch between custom `ISwapRouter` interface and actual Uniswap V3 Router implementation. When ExactInputSingleParams is initialized without explicit deadline assignment, it defaults the deadline field to 0, which is always in the past and causes Uniswap's checkDeadline modifier to revert all transactions.

IMPACT:

- All swapTokenForOraToken() calls in Swap.sol will revert
- All execute() calls in BuyAndBurn.sol will revert

- Core protocol swap operations become completely unusable

RECOMMENDATIONS:

Update the ISwapRouter interface to include the deadline parameter and set it to block.timestamp in all swap operations to ensure immediate validity.

3.2 Inconsistent Fee Calculation in executeBuyExactOut Function

SEVERITY: HIGH **STATUS:** Fixed

PATH:

contracts/logic/SmartFunLogic.sol

DESCRIPTION:

The SmartFunLogic library has a critical error in the fee calculation order in the executeBuyExactOut function. The protocol calculates fees before finalizing the transaction amount.

```
function executeBuyExactOut(
    // ... parameters
) internal pure returns (DataTypes.BuyResult memory result) {
    if (_virtualTokenReserves <= _tokenAmount) revert
        SmartFunErrors.InsufficientTokenReserves();
    uint256 collateralToSpend = (_tokenAmount *
        _virtualCollateralReserves) / (_virtualTokenReserves -
        _tokenAmount);

    // Error: Fees are calculated based on initial collateralToSpend
        estimate
    (result.helioFee, result.dexFee) = calculateFee(collateralToSpend,
        _feeBasisPoints, _dexFeeBasisPoints);

    // Then collateralToSpend may be completely recalculated here
    (_tokenAmount, collateralToSpend) = calculateTokenAmount(
        _tokenAmount,
        collateralToSpend,
        // ...other parameters
    );

    // Using old fees with new amounts results in inconsistent total
        payment
    result.collateralToPayWithFee = collateralToSpend + result.helioFee +
        result.dexFee;
    result.tokensOut = _tokenAmount;
}
```

Root Cause: Calculation order error - fee calculation occurs before the calculateTokenAmount function call which may adjust transaction amounts.

IMPACT:

Users may pay fees that don't match the actual transaction amount. When the calculateTokenAmount function reduces the token purchase amount and collateral spent due to market cap limits, users pay excessive fees.

RECOMMENDATIONS:

Recalculate fees after amount adjustments:

```
(_tokenAmount, collateralToSpend) = calculateTokenAmount(...);
// Calculate fees after all amounts are finalized
(result.helioFee, result.dexFee) = calculateFee(collateralToSpend,
    _feeBasisPoints, _dexFeeBasisPoints);
```

3.3 CometFactory lacks access control allowing complete system DOS attack via preemptive deployment

SEVERITY: HIGH **STATUS:** Fixed

PATH:

`contracts/lending/CometFactory.sol`

DESCRIPTION:

The CometFactory.deployComet() function lacks any access control and uses predictable CREATE2 salt generation, allowing attackers to completely disable the entire SmartFun migration system.

```
// contracts/lending/CometFactory.sol - No access control
function deployComet(CometConfiguration.Configuration calldata config)
    external returns (address cometAddress) {
     bytes32 salt = keccak256(abi.encode(config)); // Predictable salt
     bytes memory initCode = getInitCode(config);
     assembly ("memory-safe") {
         cometAddress := create2(0, add(initCode, 0x20), mload(initCode),
             salt)
     }
     return cometAddress;
}
```

Root Cause: 1. CometFactory.deployComet() has no access control - anyone can call it 2. All Smart Tokens use the same global _configuration from SmartFunFactory 3. CREATE2 salt is deterministic: keccak256(abi.encode(config)) 4. Global configuration is publicly readable via getCometConfiguration()

IMPACT:

- Complete system denial of service: Entire SmartFun migration functionality becomes permanently unusable
- All Smart Tokens affected: Not limited to specific tokens
- Permanent damage: Cannot be reversed without contract redeployment

RECOMMENDATIONS:

1. Add access control to CometFactory with authorized caller mapping
2. Use dynamic salt generation to prevent collisions

3.4 marginSell and closeMarginSell lack pause mechanism checks allowing continued operation during emergency pauses

SEVERITY: HIGH **STATUS:** Fixed

PATH:

contracts/lending/Comet.sol

DESCRIPTION:

The marginSell() and closeMarginSell() functions lack pause mechanism validation, allowing users to continue margin trading operations even when the system administrator has paused this functionality.

```
function marginSell(
    address borrower,
    address collateralToken,
    uint128 collateralAmount,
    uint256 borrowAmount,
    address swapRouter,
    uint24 fee,
    address tokenOut,
    uint256 minTokenOut
) override external nonReentrant returns (uint256 amountOut) {
    if (borrower == address(0)) revert InvalidAddress();
    if (swapRouter == address(0) || tokenOut == address(0)) revert
        InvalidAddress();
    if (collateralAmount == 0 || borrowAmount == 0) revert Absurd();
    // MISSING: if (isMarginSellPaused()) revert Paused();

    supplyCollateral(msg.sender, borrower, collateralToken,
        collateralAmount);
    _withdrawBase(borrower, borrower, borrowAmount, false);
    amountOut = _swapTokenForTokenOut(baseToken, borrowAmount,
        swapRouter, fee, tokenOut, minTokenOut);
    doTransferOut(tokenOut, borrower, amountOut);
    emit MarginSell(borrower, collateralToken, collateralAmount,
        borrowAmount, tokenOut, amountOut);
}
```

Root Cause: marginSell() and closeMarginSell() functions missing isMarginSellPaused() checks.

IMPACT:

- Emergency response failure: pause mechanism bypassed during security incidents
- System risk amplification: Margin trading continues during crisis situations

RECOMMENDATIONS:

Add pause checks to both margin trading functions:

```
if (isMarginSellPaused()) revert Paused();
```

3.5 Permanently locks ETH through unrestricted receive()

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

contracts/protocol/BuyAndBurn.sol

DESCRIPTION:

BuyAndBurn contract accepts ETH via receive() function but lacks withdrawal mechanism. Direct ETH transfers result in permanent lockup.

```
// contracts/protocol/BuyAndBurn.sol
receive() external payable {}
```

Root Cause: No access control or emergency withdrawal functions implemented.

IMPACT:

Direct ETH transfers to contract address cause permanent fund loss with no recovery mechanism.

RECOMMENDATIONS:

Remove receive() function or add emergency withdrawal.

3.6 Hard-coded fee tier in UniswapV3 swap operations

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

contracts/protocol/Swap.sol

DESCRIPTION:

Multiple contracts use hard-coded fee tiers (10000 = 1%) for UniswapV3 swaps, preventing optimal liquidity utilization across different fee tiers.

```
// contracts/protocol/Swap.sol
uint24 public constant POOL_FEE = 10000;

function swapTokenForOraToken(
    address tokenIn,
    uint256 amountIn,
    address receiver
) external payable nonReentrant returns (uint256 amountOut) {
    address pool = IV3Factory(v3Factory).getPool(tokenIn, oraToken,
        POOL_FEE);
    require(pool != address(0), "Swap: Pool not exists");

    ISwapRouter.ExactInputSingleParams memory params =
        ISwapRouter.ExactInputSingleParams({
         tokenIn: tokenIn,
         tokenOut: oraToken,
         fee: POOL_FEE,  // Hard-coded fee tier
         recipient: receiver,
         amountIn: amountIn,
         amountOutMinimum: amountOutMin,
         sqrtPriceLimitX96: 0
    });
}
```

Root Cause: Fixed fee tier prevents utilization of better liquidity available at different fee tiers (0.05%, 0.3%).

IMPACT:

The hard-coded fee tier may not exist for that token pair, causing the swap to fail, even though liquidity does exist at a different fee tier. Additionally, the hard-coded fee tier may provide inferior liquidity resulting in greater slippage for the user.

RECOMMENDATIONS:

Implement fee tier selection logic or allow user-specified fee tiers.

3.7 Unchecked ETH acceptance in non-ETH token operations leads to permanent fund lockup

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

contracts/protocol/BuyAndBurn.sol

DESCRIPTION:

Multiple payable functions in the protocol fail to properly handle ETH when the input token is not ETH/WETH, leading to potential ETH lockup and fund loss.

```
function _swapTokenForOraToken(address _tokenIn, uint256 _amountIn)
    private returns (uint256 amountOut) {
     if (_tokenIn == address(oraToken)) {
         // No check for msg.value when tokenIn is ORA
         oraToken.safeTransferFrom(msg.sender, address(this), _amountIn);
         amountOut = _amountIn;
     } else if (_tokenIn == address(0)) {
         if (msg.value < _amountIn) revert
             SmartFunErrors.InsufficientETH();
         amountOut = swap.swapTokenForOraToken{value:
             _amountIn}(swap.WETH(), _amountIn, address(this));
         if (msg.value > _amountIn) {
             (bool sent,) = payable(msg.sender).call{value: msg.value -
                 _amountIn}("");
             require(sent, "Refund failed");
         }
     } else {
         // No check for msg.value when tokenIn is ERC20
         IERC20(_tokenIn).safeTransferFrom(msg.sender, address(swap),
             _amountIn);
         amountOut = swap.swapTokenForOraToken(_tokenIn, _amountIn,
             address(this));
     }
}
```

Root Cause: Functions marked as `payable` accept ETH but only handle it when the input token is ETH/WETH.

IMPACT:

- Users may accidentally send ETH when using ERC20 tokens, resulting in permanent fund loss
- No mechanism exists to recover accidentally sent ETH

RECOMMENDATIONS:

Reject excess ETH in non-ETH operations by adding msg.value validation or implementing automatic refund mechanism to prevent fund lockup.

3.8 ERC20::approve will revert to some non-standard tokens like USDT

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

contracts/protocol/Swap.sol

DESCRIPTION:

Multiple contracts use direct approve() calls without considering non-standard ERC20 tokens like USDT that have different approval behavior.

```
function swapTokenForOraToken(
    address tokenIn,
    uint256 amountIn,
    address receiver
) external payable nonReentrant returns (uint256 amountOut) {
    // ...
    if (tokenIn == WETH) {
        return ISwapRouter(v3Router).exactInputSingle{value:
            amountIn}(params);
    } else {
        uint256 currentAllowance =
            ERC20(tokenIn).allowance(address(this), v3Router);
        if (currentAllowance < amountIn) {
            ERC20(tokenIn).approve(v3Router, type(uint256).max); //
                Vulnerable to non-standard tokens
        }
        return ISwapRouter(v3Router).exactInputSingle(params);
    }
}
```

Root Cause: Direct use of `approve()` method without considering non-standard ERC20 tokens that may have different function signatures or approval logic.

IMPACT:

Core protocol functions (swapping, LP fee distribution, buy-and-burn) will fail when interacting with non-standard ERC20 tokens like USDT, USDC, or other tokens with custom approval behavior.

RECOMMENDATIONS:

Use SafeERC20.forceApprove() instead of direct approve() calls to handle all ERC20 token variants.

3.9 Token approval with type(uint256).max causes incompatibility with certain ERC20 tokens

SEVERITY: MEDIUM

STATUS: Fixed

PATH:

`contracts/protocol/Swap.sol`

DESCRIPTION:

The protocol uses type(uint256).max for token approvals, which is incompatible with tokens that reject unlimited approvals (like UNI, COMP), causing swap operations to revert.

```
// contracts/protocol/Swap.sol
ERC20(tokenIn).approve(v3Router, type(uint256).max);

// contracts/protocol/BuyAndBurn.sol
oraToken.approve(address(swapRouter), type(uint256).max);
```

Root Cause: Some ERC20 tokens explicitly reject `type(uint256).max` approvals for security reasons.

IMPACT:

Swap operations fail for incompatible tokens.

RECOMMENDATIONS:

Use exact amount approvals instead of type(uint256).max.

3.10 swapTokenForOraToken lacks msg.value validation

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

contracts/protocol/Swap.sol

DESCRIPTION:

The swapTokenForOraToken function is external and can be called directly by users, but lacks msg.value validation when tokenIn is WETH.

```
function swapTokenForOraToken(
    address tokenIn,
    uint256 amountIn,
    address receiver
) external payable nonReentrant returns (uint256 amountOut) {
    // ...
    if (tokenIn == WETH) {
        return ISwapRouter(v3Router).exactInputSingle{value:
            amountIn}(params); // No msg.value check
    } else {
        uint256 currentAllowance =
            ERC20(tokenIn).allowance(address(this), v3Router);
        if (currentAllowance < amountIn) {
             ERC20(tokenIn).approve(v3Router, type(uint256).max);
        }
        return ISwapRouter(v3Router).exactInputSingle(params);
    }
}
```

Root Cause: Function accepts `msg.value` but doesn't validate it equals `amountIn` for WETH swaps.

IMPACT:

- Excess ETH permanently locked in contract when users send more than amountIn
- Transaction failures when insufficient ETH provided

RECOMMENDATIONS:

Add explicit msg.value validation.

3.11 swapTokenForOraToken lacks token transfer handling

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

contracts/protocol/Swap.sol

DESCRIPTION:

The swapTokenForOraToken function is external and can be called directly by users, but doesn't transfer ERC20 tokens from users when tokenIn is not WETH.

```
function swapTokenForOraToken(
    address tokenIn,
    uint256 amountIn,
    address receiver
) external payable nonReentrant returns (uint256 amountOut) {
    // ... validation logic
    if (tokenIn == WETH) {
        return ISwapRouter(v3Router).exactInputSingle{value:
            amountIn}(params);
    } else {
        // No token transfer from user - expects pre-transferred tokens
        uint256 currentAllowance =
            ERC20(tokenIn).allowance(address(this), v3Router);
        if (currentAllowance < amountIn) {
             ERC20(tokenIn).approve(v3Router, type(uint256).max);
        }
        return ISwapRouter(v3Router).exactInputSingle(params);
    }
}
```

Root Cause: Function assumes tokens are already transferred to contract but doesn't handle the transfer itself.

IMPACT:

- Direct function calls fail when users don't pre-transfer tokens

- Inconsistent behavior between direct calls and wrapper contract calls

RECOMMENDATIONS:

Add token transfer logic or restrict function access.

3.12 Market cap calculation parameter inconsistency leads to incorrect user decisions

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

contracts/protocol/SmartFunFactory.sol

DESCRIPTION:

estimateTokenAmount function uses initialTokenSupply for market cap calculation while the actual SmartToken.getMarketCap() function uses totalSupply().

```
// contracts/protocol/SmartFunFactory.sol
uint256 marketCap = SmartFunLogic.getMarketCap(newVirtualCollateral,
    newVirtualToken, initialTokenSupplyAmount);

// contracts/protocol/SmartToken.sol
function getMarketCap() public view returns (uint256) {
    return
        SmartFunLogic.getMarketCap(_tokenState.virtualCollateralReserves,
        _tokenState.virtualTokenReserves, totalSupply());
}
```

Root Cause: The Factory's market cap estimation uses the fixed `initialTokenSupply` parameter, while the actual market cap calculation uses the dynamic `totalSupply()` which decreases after burn operations.

IMPACT:

- Users receive incorrect market cap estimates from factory functions
- Market cap threshold checks in estimateTokenAmount use incorrect values

RECOMMENDATIONS:

Update the Factory's estimateTokenAmount function to use totalSupply() instead of initialTokenSupply for market cap calculations.

3.13 TokenConfig lacks validation allowing duplicate tokens and ID mismatch

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

contracts/protocol/SmartFunFactory.sol

DESCRIPTION:

_createNewToken doesn't validate TokenConfig parameters, allowing users to create duplicate tokens with identical configurations and arbitrary IDs.

```
function _createNewToken(
    DataTypes.TokenConfig calldata _tokenConfig,
    bytes32 _salt
) internal returns (address tokenAddress) {
    uint256 tokenId = currentTokenId + 1;
    bytes memory initCode = getInitCode(_tokenConfig);

    // No validation of _tokenConfig parameters
    // No check for duplicate configurations
    // No validation that _tokenConfig.ID == tokenId

    assembly ("memory-safe") {
        tokenAddress := create2(0, add(initCode, 0x20), mload(initCode),
            _salt)
    }
    if (tokenAddress == address(0)) revert
        SmartFunErrors.InvalidTokenAddress();
    if (deployed[tokenAddress]) revert SmartFunErrors.AlreadyDeployed();
    deployed[tokenAddress] = true;
    currentTokenId = tokenId;
}
```

Root Cause: CREATE2 address depends on `initCode` + `salt`, so different salts with identical `TokenConfig` create different addresses.

IMPACT:

- Users can create multiple tokens with identical names, symbols, and descriptions
- Token IDs don't match factory's sequential numbering, causing data inconsistency

RECOMMENDATIONS:

Add validation to ensure _tokenConfig.ID == currentTokenId + 1 and implement duplicate configuration checks using a mapping of configuration hashes.

3.14 CometRouter Lack of Liquidator Authorization Check

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

contracts/lending/CometRouter.sol

DESCRIPTION:

The _requireAuthorizedLiquidator function in CometRouter contract only checks its own isLiquidator mapping, ignoring the independent liquidator permission management in Comet contract.

```
// contracts/lending/CometRouter.sol
function _requireAuthorizedLiquidator(IComet cometContract) internal view
    {
    require(isLiquidator[msg.sender] || owner() == msg.sender ||
        cometContract.governor() == msg.sender, ”Router: Unauthorized”);
}
```

Root Cause: CometRouter and Comet contracts maintain independent liquidator mappings managed by different roles.

IMPACT:

Authorized liquidators in Comet protocol cannot execute liquidation operations through CometRouter, affecting the normal operation of the liquidation system.

RECOMMENDATIONS:

Add verification of Comet liquidator status in the authorization check:

```
require(
    isLiquidator[msg.sender] ||
    owner() == msg.sender ||
    cometContract.governor() == msg.sender ||
    cometContract.isLiquidator(msg.sender),
    ”Router: Unauthorized”
);
```

3.15 Transfer Restriction Bypass via transferFrom

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

contracts/protocol/SmartToken.sol

DESCRIPTION:

The SmartToken contract implements transfer restrictions in the transfer function but fails to apply the same restrictions to transferFrom operations.

```
// SmartToken.sol - Transfer function with restrictions:
function transfer(address _to, uint256 _value) public override(ERC20,
    IERC20) returns (bool) {
     if ((_to == _addresses.pair || _to == address(this)) &&
         !_tokenState.sendingToPairAllowed)
         revert SmartFunErrors.SendingToPairIsNotAllowedBeforeMigration();
     return super.transfer(_to, _value);
}

// Missing transferFrom override - No such function exists in
    SmartToken.sol
```

Root Cause: 1. Only `transfer` function is overridden with restrictions 2. `transferFrom` uses default ERC20 implementation which bypasses restriction checks 3. Internal `_transfer` function doesn't include the restriction checks

IMPACT:

1. Bypassing Transfer Restrictions: Users can transfer tokens to restricted addresses before migration using `approve` + `transferFrom`
2. Off-chain OTC Trading: The restriction bypass enables off-chain OTC trading that affects the bonding curve's price discovery mechanism

RECOMMENDATIONS:

Override transferFrom Function with the same restrictions:

```
function transferFrom(address from, address to, uint256 amount) public
    override returns (bool) {
     if ((to == _addresses.pair || to == address(this)) &&
         !_tokenState.sendingToPairAllowed)
          revert SmartFunErrors.SendingToPairIsNotAllowedBeforeMigration();
     return super.transferFrom(from, to, amount);
}
```

3.16 Unrestricted Fee Percentage Setting

SEVERITY: LOW **STATUS:** Acknowledge

PATH:

contracts/protocol/LPFeeDistributor.sol

DESCRIPTION:

The LPFeeDistributor.sol contract grants the owner the ability to arbitrarily modify the fee percentage, which introduces a centralization risk.

```
function setFeePercentage(uint256 _newFeePercentage) external onlyOwner {
    require(_newFeePercentage <= 100, "LPFeeDistributor: The maximum
        percentage of fee is 100");
    feePercentage = _newFeePercentage;
    emit FeePercentageUpdated(_newFeePercentage);
}
```

Root Cause: While a maximum cap of 100% is enforced for _newFeePercentage, the owner can set the feePercentage to any value up to this maximum without a time-lock or additional governance.

IMPACT:

- Abuse of Fee Percentage: A malicious owner could set the fee percentage to an unreasonably high value (e.g., 100%), effectively confiscating all or a significant portion of user earnings from LP fees.
- Centralization Risk: The onlyOwner modifier creates a single point of failure and allows for instant, unchecked changes to core protocol economics.

RECOMMENDATIONS:

1. Implement a Reasonable Fee Percentage Cap: Reduce the hardcoded maximum for _newFeePercentage to a more reasonable and protocol-friendly value (e.g., 10-25%).
2. Add a Time-lock for Fee Percentage Changes: Introduce a time-lock mechanism for the setFeePercentage function.

3.17 LPFeeDistributor lacks slippage protection in swap operations

SEVERITY: LOW **STATUS:** Acknowledge

PATH:

`contracts/protocol/LPFeeDistributor.sol`

DESCRIPTION:

LPFeeDistributor performs swaps without slippage protection, making it vulnerable to price manipulation attacks. The amountOutMinimum parameter is hardcoded to 0, allowing unlimited slippage.

```
function _swapExactInputSingle(
    address tokenIn,
    address tokenOut,
    uint256 amountIn,
    uint24 poolFeeVal
) private {
    ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
        .ExactInputSingleParams({
        tokenIn: tokenIn,
        tokenOut: tokenOut,
        fee: poolFeeVal,
        recipient: address(this),
        amountIn: amountIn,
        amountOutMinimum: 0,  // No slippage protection
        sqrtPriceLimitX96: 0
    });

    swapRouter.exactInputSingle(params);
}
```

IMPACT:

Vulnerable to price manipulation attacks due to unlimited slippage tolerance.

RECOMMENDATIONS:

Add slippage protection by calculating minimum output amount based on expected price.

3.18 Asset count limit inconsistency between design capacity and deployment constraint

SEVERITY: LOW **STATUS:** Acknowledge

PATH:

`contracts/lending/CometCore.sol`

DESCRIPTION:

An inconsistency between the system's design capacity and deployment limitations regarding the maximum number of supported assets.

```
// contracts/lending/CometCore.sol
uint8 internal constant MAX_ASSETS = 24;

// contracts/lending/Comet.sol
uint8 internal constant MAX_ASSETS_FOR_ASSET_LIST = 5;

if (config.assetConfigs.length > MAX_ASSETS_FOR_ASSET_LIST) revert
    TooManyAssets();
```

Root Cause: CometCore.sol defines MAX_ASSETS = 24 with supporting bit vector infrastructure while Comet.sol restricts deployment to MAX_ASSETS_FOR_ASSET_LIST = 5 assets.

IMPACT:

1. Future Expansion Blocked: Current deployment cannot utilize the full designed capacity without contract upgrades
2. Integration Complexity: External systems may assume 24-asset capacity based on `maxAssets()` function

RECOMMENDATIONS:

Align the deployment constraint with the design capacity or reduce design limit to match current needs.

3.19 Swap router validation missing in margin trading functions

SEVERITY: LOW **STATUS:** Acknowledge

PATH:

contracts/lending/Comet.sol

DESCRIPTION:

The marginSell() and closeMarginSell() functions lack swap router validation, allowing users to specify arbitrary router addresses.

```
function marginSell(
    address borrower,
    address collateralToken,
    uint128 collateralAmount,
    uint256 borrowAmount,
    address swapRouter,
    uint24 fee,
    address tokenOut,
    uint256 minTokenOut
) override external nonReentrant returns (uint256 amountOut) {
    if (swapRouter == address(0) || tokenOut == address(0)) revert
        InvalidAddress();
    // Missing: router whitelist validation

    supplyCollateral(msg.sender, borrower, collateralToken,
        collateralAmount);
    _withdrawBase(borrower, borrower, borrowAmount, false);
    amountOut = _swapTokenForTokenOut(baseToken, borrowAmount,
        swapRouter, fee, tokenOut, minTokenOut);
    doTransferOut(tokenOut, borrower, amountOut);
}
```

IMPACT:

Malicious routers can steal approved tokens or manipulate swap outcomes.

RECOMMENDATIONS:

Add router whitelist validation in core Comet contract.

3.20 Variable naming is inconsistent

SEVERITY: INFO **STATUS:** Fixed

PATH:

contracts/logic/SmartFunLogic.sol

DESCRIPTION:

In SmartFunLogic.sol, the variable collaterallToReceive uses an inconsistent naming style (should be collateralToReceive).

```
// contracts/logic/SmartFunLogic.sol
uint256 collaterallToReceive = (_tokenAmount *
    _virtualCollateralReserves) / (_virtualTokenReserves + _tokenAmount);
(result.helioFee, result.dexFee) = calculateFee(collaterallToReceive,
    _feeBasisPoints, _dexFeeBasisPoints);
result.collateralToReceiveMinusFee = collaterallToReceive -
    result.helioFee - result.dexFee;
```

IMPACT:

Code clarity and consistency issues.

RECOMMENDATIONS:

Rename collaterallToReceive to collateralToReceive for clarity and consistency.

3.21 Migration state not reset after successful migration

SEVERITY: INFO **STATUS:** Fixed

PATH:

contracts/protocol/SmartFunFactory.sol

DESCRIPTION:

The migrate() function does not reset the readyForMigration state after successful migration, allowing repeated migration attempts.

```
function migrate(address _token) external nonReentrant whenNotPaused {
    if (!deployed[_token]) revert SmartFunErrors.TokenNotWhitelisted();
    if (!readyForMigration[_token]) revert
        SmartFunErrors.NotReadyForMigration();

    ISmartToken(_token).deployComet(_addressConfig.cometFactory,
        _configuration);
    // ... migration logic

    emit SmartFunEvents.Migrated(_token, /* ... */);
    // Missing: readyForMigration[_token] = false;
}
```

IMPACT:

State inconsistency allowing repeated migration attempts.

RECOMMENDATIONS:

Reset the migration state after successful migration.

3.22 ETH address check is non-standard

SEVERITY: INFO **STATUS:** Fixed

PATH:

`Multiple files`

DESCRIPTION:

ETH is identified only by address(0), not by the EIP-7528 standard address 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeee

```
if (_tokenIn == address(0)) { ... }
```

This may cause compatibility issues with wallets and aggregators that use the EIP-7528 standard.

IMPACT:

Compatibility issues with wallets and aggregators that use EIP-7528 standard.

RECOMMENDATIONS:

Check for both representations:

```
if (_tokenIn == 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeeEEeE || _tokenIn ==
    address(0)) { ... }
```

3.23 LP Fee Calculation Bypasses Uniswap V3's Accurate Values

SEVERITY: INFO **STATUS:** Fixed

PATH:

contracts/protocol/LPFeeDistributor.sol

DESCRIPTION:

getLpFeesByTokenId() manually calculates fees instead of using Uniswap V3's built-in tokensOwed0 and tokensOwed1 values.

```
function getLpFeesByTokenId(uint256 tokenId) public view returns (address
    token0, address token1, uint128 amount0, uint128 amount1){
    // Skips the accurate tokensOwed0 and tokensOwed1 values
    (,, token0, token1, fee,,, liquidity, feeGrowthInside0LastX128,
        feeGrowthInside1LastX128,,) = positionManager.positions(tokenId);

    // Manual calculation that ignores tick boundaries
    uint256 feeGrowthInside0X128 = feeGrowthGlobal0X128 -
        feeGrowthInside0LastX128;
    uint256 feeGrowthInside1X128 = feeGrowthGlobal1X128 -
        feeGrowthInside1LastX128;

    amount0 = uint128((uint256(liquidity) * feeGrowthInside0X128) / (2 **
        128));
    amount1 = uint128((uint256(liquidity) * feeGrowthInside1X128) / (2 **
        128));
}
```

Root Cause: Ignores Uniswap V3's complex fee calculation that handles tick boundaries and position range status.

IMPACT:

Inaccurate fee calculations that may not match actual Uniswap V3 values.

RECOMMENDATIONS:

Use Uniswap V3's accurate values from the position manager.

3.24 Price oracle precision mismatch

SEVERITY: INFO **STATUS:** Fixed

PATH:

contracts/lending/AssetList.sol

DESCRIPTION:

The claim that there is a precision mismatch between CometCore.sol (18 decimals) and AssetList.sol (8 decimals) is incorrect. The PRICE_FEED_DECIMALS = 8 constant in AssetList.sol is never used in the codebase.

```
// contracts/lending/AssetList.sol
uint8 internal constant PRICE_FEED_DECIMALS = 8;  // Defined but never
   used

// contracts/lending/CometCore.sol
uint8 internal constant PRICE_FEED_DECIMALS = 18;  // Actually used
uint64 internal constant PRICE_SCALE = uint64(10 ** PRICE_FEED_DECIMALS);
    // = 1e18
```

IMPACT:

Code clarity issue with unused constants that could cause confusion.

RECOMMENDATIONS:

Remove the unused PRICE_FEED_DECIMALS = 8 constant from AssetList.sol to avoid confusion.

4. CONCLUSION

In this audit, we thoroughly analyzed **Smart Fun** smart contract implementation. The problems found are described and explained in detail in Section 3. The problems found in the audit have been communicated to the project leader. We therefore consider the audit result to be **PASSED**.

To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5. APPENDIX

5.1 Basic Coding Assessment

5.1.1 Apply Verification Control

Description	The security of apply verification
Result	Not found
Severity	CRITICAL

5.1.2 Authorization Access Control

Description	Permission checks for external integral functions
Result	Not found
Severity	CRITICAL

5.1.3 Forged Transfer Vulnerability

Description	Assess whether there is a forged transfer notification vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.4 Transaction Rollback Attack

Description	Assess whether there is transaction rollback attack vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.5 Transaction Block Stuffing Attack

Description	Assess whether there is transaction blocking attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.6 Soft Fail Attack Assessment

Description	Assess whether there is soft fail attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.7 Hard Fail Attack Assessment

Description	Examine for hard fail attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.8 Abnormal Memo Assessment

Description	Assess whether there is abnormal memo vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.9 Abnormal Resource Consumption

Description	Examine whether abnormal resource consumption in contract processing
Result	Not found
Severity	CRITICAL

5.1.10 Random Number Security

Description	Examine whether the code uses insecure random number
Result	Not found
Severity	CRITICAL

5.2 Advanced Code Scrutiny

5.2.1 Cryptography Security

Description	Examine for weakness in cryptograph implementation
Result	Not found
Severity	HIGH

5.2.2 Account Permission Control

Description	Examine permission control issue in the contract
Result	Not found
Severity	MEDIUM

5.2.3 Malicious Code Behavior

Description	Examine whether sensitive behavior present in the code
Result	Not found
Severity	MEDIUM

5.2.4 Sensitive Information Disclosure

Description	Examine whether sensitive information disclosure issue present in the code
Result	Not found
Severity	MEDIUM

5.2.5 System API

Description	Examine whether system API application issue present in the code
Result	Not found
Severity	LOW

6. DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without ExVul's prior written consent.

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts ExVul to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. ExVul's position is that each company and individual are responsible for their own due diligence and continuous security. ExVul's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

7. REFERENCES

[1] MITRE. CWE-191: Integer Underflow (Wrap or Wraparound). https://cwe.mitre.org/data/definitions/191.html.

[2] MITRE. CWE-197: Numeric Truncation Error. https://cwe.mitre.org/data/definitions/197.html.

[3] MITRE. CWE-400: Uncontrolled Resource Consumption. https://cwe.mitre.org/data/definitions/400.html.

[4] MITRE. CWE-440: Expected Behavior Violation. https://cwe.mitre.org/data/definitions/440.html.

[5] MITRE. CWE-684: Protection Mechanism Failure. https://cwe.mitre.org/data/definitions/693.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[7] MITRE. CWE CATEGORY: Behavioral Problems. https://cwe.mitre.org/data/definitions/438.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE CATEGORY: Resource Management Errors. https://cwe.mitre.org/data/definitions/399.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

Contact

Website
www.exvul.com

Email
contact@exvul.com

Twitter
@EXVULSEC

Github
github.com/EXVUL-Sec

ExVul