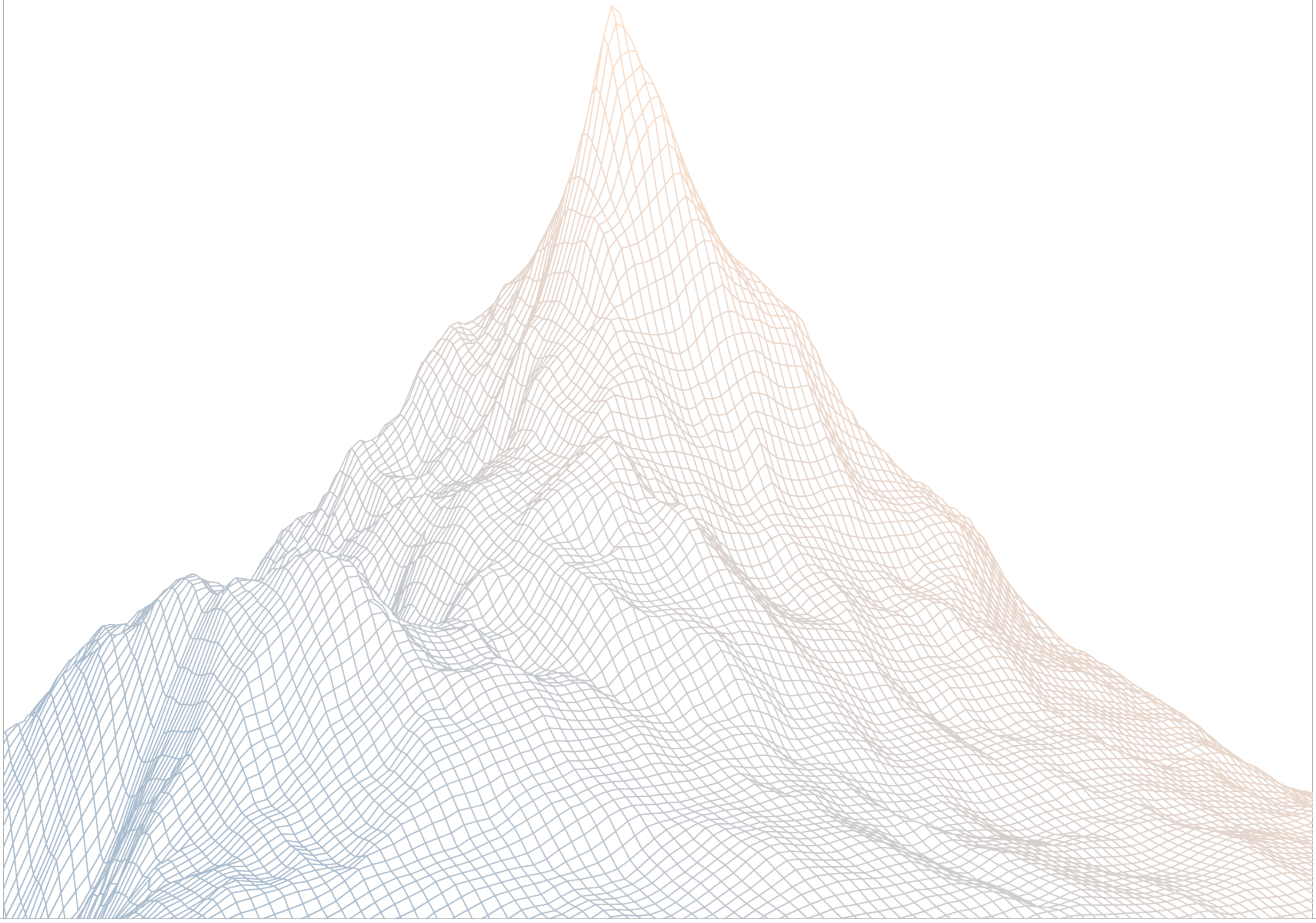
Zenith

Nad.fun

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: November 5th to November 10th, 2025

AUDITED BY:
ast3ros
sin1st3r

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About Nad.fun

Welcome to Nad.fun! We're redefining the memecoin experience by combining token creation, trading, and social engagement in one seamless platform.

2.2 Scope

The engagement involved a review of the following targets:

Target	contract-v3
Repository	https://github.com/Naddotfun/contract-v3
Commit Hash	`22c4be543ef0030a2d224021e44e998f95c25840`
Files	`src/**/*.sol`

2.3 Audit Timeline

November 5, 2025	Audit start
November 10, 2025	Audit end
November 13, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	2
High Risk	0
Medium Risk	3
Low Risk	11
Informational	4
Total Issues	**20**

3 Findings Summary

ID	Description	Status
C-1	Incorrect initial price due to wrong token ordering assumption	Resolved
C-2	Duplicate claim allows fund drainage	Resolved
M-1	exactOutSell fail to validate exact output amount	Resolved
M-2	Refund sent to recipient instead of msg.sender	Resolved
M-3	Inconsistent fee calculation leads to under collection	Resolved
L-1	Unnecessary allowance check against maximum instead of actual transfer amount	Resolved
L-2	Creator exemption from anti-sniping penalty bypassed in exactOutBuy	Resolved
L-3	exactOutBuy reverts when anti-sniping penalties are active	Resolved
L-4	Missing available token check in create function	Resolved
L-5	Event not emitted for non-penalized transactions	Resolved
L-6	Blacklisted actor can still receive fund allocation	Acknowledged
L-7	Slippage protection bypassed due to estimate-based validation	Resolved
L-8	Front-running pool creation causes DOS on token launch	Acknowledged
L-9	Residual token allowances not reset after minting	Resolved

ID	Description	Status
L-10	Reward pool incorrectly accounts for fee-on-transfer tokens	Resolved
L-11	Event emits incorrect token amount in sell function	Resolved
I-1	Misleading comment in TransferHelper	Acknowledged
I-2	Unreachable overflow check in addActor	Resolved
I-3	Missing allowance validation in sellPermit	Acknowledged
I-4	Unused ReentrancyGuard inheritance	Resolved

4 Findings

4.1 Critical Risk

A total of 2 critical risk findings were identified.

[C-1] Incorrect initial price due to wrong token ordering assumption

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- DexDeployer.sol#L61-L64

Description:

The `createPool` function incorrectly assumes WMON is always `token0` when calculating the initial price, but Uniswap V3 determines token ordering by address comparison. When the new token has an address lower than WMON, the token becomes `token0` and WMON becomes `token1`, inverting the price calculation.

It leads to price inversion, wrong aligned tick calculation, ncorrect liquidity range and liquidity drainage.

```
function createPool(address token) external returns (address pool) {
    IUniswapV3Factory _factory = IUniswapV3Factory(factory);
    pool = _factory.createPool(token, wMon, feeTier);

    //INITIALIZE
    (
        uint256 virtualMonReserve,
        uint256 virtualTokenReserve,
        uint256 targetVirtualTokenAmount
    ) = IBondingCurve(curve).config();
    uint256 k = virtualMonReserve * virtualTokenReserve;
    uint256 targetVirtualMonAmount = k / targetVirtualTokenAmount;

    uint160 sqrtPrice = _calculateSqrtPrice(
        targetVirtualMonAmount,
        targetVirtualTokenAmount
    );

    IUniswapV3Pool(pool).initialize(sqrtPrice);
    /// The cardinality required by Ammplify for twap prices.
    IUniswapV3Pool(pool).increaseObservationCardinalityNext(32);

    return pool;
}
```

Consider a scenario:

At graduation, targetVirtualMonAmount = 345015903150991418937616, targetVirtualTokenAmount = 279900191000000000000000000. When `token address < WMON address`:

- Price inversion: Intended price of `targetVirtualMonAmount/targetVirtualTokenAmount` (345e18/279e24 ≈ 0.00123 WMON per token) becomes inverted to ~811 WMON per token
- Wrong tick calculation: Aligned tick becomes `66800` instead of `-66800`
- Incorrect liquidity range: WMON liquidity is provided from tick `-94200 to 66600` (from bondingTick to aligged tick - Tickspacing) instead of the correct range (`-94200 to -66600`). It provides WMON at very cheap price.
- Liquidity drainage: Attackers can swap 10e18 tokens for ~7495e18 WMON, draining the pool at a massive discount

Recommendations:

Determine `token0/token1` ordering before calculating the initial price.

Nad.fun: Resolved with `@e52ad0fb4e`.

Zenith: Verified.

[C-2] Duplicate claim allows fund drainage

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- CreatorTreasury.sol#L205-L208
- CreatorTreasury.sol#L219-L221

Description:

The `claim` function updates the `claimed` mapping only after the verification loop completes. This allows a malicious creator to include duplicate entries in a single batch, bypassing the double-claim check.

We can see that the duplicate check `claimed[checkLeaf]` occurs before any claimed flags are set, enabling the same leaf to pass validation multiple times in one transaction.

```
function claim(address[] calldata tokens, uint256[] calldata amounts,
    bytes32[][] calldata merkleProofs)
    external
    nonReentrant
{
    ...
    for (uint256 i = 0; i < length; i++) {
        address token = tokens[i];
        uint256 amount = amounts[i];
        bytes32[] calldata merkleProof = merkleProofs[i];

        // 3a. Verify msg.sender is creator through CreatorManager
        address creator = ICreatorManager(creatorManager).creators(token);
        if (creator != msg.sender) {
            revert InvalidMerkleProof();
        }

        // 3b. Use current merkleRoot (all tokens use the same merkleRoot)

        // 3c. Generate leaf for Merkle proof verification
        bytes32 leaf = keccak256(abi.encode(token, creator, amount));
```

```
            // 3d. Generate checkLeaf for duplicate claim checking
            bytes32 checkLeaf = keccak256(abi.encode(token, creator, amount,
        merkleRoot));
            checkLeaves[i] = checkLeaf;

            // 3e. Check for duplicate claims within the same merkleRoot period
            if (claimed[checkLeaf]) {
                revert AlreadyClaimed();
            }

            // 3f. Verify Merkle proof
            if (!MerkleProof.verify(merkleProof, merkleRoot, leaf)) {
                revert InvalidMerkleProof();
            }
        }

        // ========================================
        // 4. State update (Effects)
        // ========================================
        for (uint256 i = 0; i < length; i++) {
            claimed[checkLeaves[i]] = true;
        }

        // ========================================
        // 5. wMon unwrap and transfer (Interactions)
        // ========================================
        // Unwrap wMon to obtain Native tokens
        IWMon(wMon).withdraw(totalAmount);

        // Safely transfer Native tokens to creator
        TransferHelper.safeTransferMon(msg.sender, totalAmount);

        // ========================================
        // 6. Emit events
        // ========================================
        for (uint256 i = 0; i < length; i++) {
            emit Claim(tokens[i], msg.sender, amounts[i]);
        }
    }
```

A creator could submit:

- tokens = [T, T]
- amounts = [A, A]
- merkleProofs = [P, P]

Both entries pass the `claimed[checkLeaf]` check since neither is marked as claimed yet.

The contract sums `totalAmount = 2A` and transfers double the intended amount, but only marks the leaf as claimed once.

Impact: A creator can drain the entire treasury by repeatedly claiming the same allocation multiple times in a single transaction.

Recommendations:

Update the claimed mapping immediately after each entry is verified, within the verification loop. This prevents duplicate entries from passing validation within the same transaction.

Nad.fun: Resolved with `@44e7d144de ...`.

Zenith: Verified.

4.2 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] exactOutSell fail to validate exact output amount

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- DexRouter.sol#L403
- DexRouter.sol#L466

Description:

The `exactOutSell` and `exactOutSellPermit` functions do not verify that the actual swap output matches the requested amount. When pool liquidity is insufficient, the Uniswap V3 swap may return less than `necessaryAmountOut`, but the function proceeds without validation, violating the "exact output" guarantee. It will also charge the fee that was computed from the requested amount, not the actually received amount.

```
    function exactOutSell(ExactOutSellParams calldata params)
        external
        nonReentrant
        ensure(params.deadline)
        returns (uint256 amountOut)
    {
         ...
        uint256 feeAmount = calculateFeeAmount(params.amountOut);
>>>     uint256 necessaryAmountOut = params.amountOut + feeAmount;

        // 4. Exact Output (negative)
        int256 amountSpecified = -int256(necessaryAmountOut);

        // 5. Execute swap (recipient address is router)
        int256 amount0;
        int256 amount1;
        {
            uint160 sqrtPriceLimitX96 =
                dexData.zeroForOne ? (TickMath.MIN_SQRT_RATIO + 1) :
    (TickMath.MAX_SQRT_RATIO - 1);
            (amount0, amount1) = IUniswapV3Pool(dexData.pool).swap(
                address(this), dexData.zeroForOne, amountSpecified,
    sqrtPriceLimitX96, data
            );
        }

        // 6. Received wMON (should be exactly necessaryAmountOut)
>>>     uint256 totalAmountOut = uint256(-(dexData.zeroForOne ? amount1 :
    amount0));

        // 7. Send fee and calculate net amount
        sendFee(feeAmount);
        amountOut = totalAmountOut - feeAmount;

        ...
    }
```

Reference Implementation (Uniswap V3 Router):

```
function exactOutputInternal(
    uint256 amountOut,
    address recipient,
    uint160 sqrtPriceLimitX96,
    SwapCallbackData memory data
) private returns (uint256 amountIn) {
    // allow swapping to the router address with address 0
    if (recipient == address(0)) recipient = address(this);

    (address tokenOut, address tokenIn, uint24 fee)
    = data.path.decodeFirstPool();

    bool zeroForOne = tokenIn < tokenOut;

    (int256 amount0Delta, int256 amount1Delta) =
        getPool(tokenIn, tokenOut, fee).swap(
            recipient,
            zeroForOne,
            -amountOut.toInt256(),
            sqrtPriceLimitX96 == 0
                ? (zeroForOne ? TickMath.MIN_SQRT_RATIO + 1 :
    TickMath.MAX_SQRT_RATIO - 1)
                : sqrtPriceLimitX96,
            abi.encode(data)
        );

    uint256 amountOutReceived;
    (amountIn, amountOutReceived) = zeroForOne
        ? (uint256(amount0Delta), uint256(-amount1Delta))
        : (uint256(amount1Delta), uint256(-amount0Delta));
    // it's technically possible to not receive the full output amount,
    // so if no price limit has been specified, require this possibility away
    if (sqrtPriceLimitX96 == 0) require(amountOutReceived == amountOut);
}
```

Recommendations:

Add validation to ensure the exact output guarantee.

Nad.fun: Resolved with `@f53e362e20 ...`.

Zenith: Verified.

[M-2] Refund sent to recipient instead of msg.sender

SEVERITY: Medium	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: High

Target

- DexRouter.sol#L238
- DexRouter.sol#L415

Description:

The `DexRouter.exactOutBuy` function incorrectly refunds excess ETH to `params.to` (the token recipient) instead of `msg.sender` (the caller who provided the ETH).

```
function exactOutBuy(ExactOutBuyParams calldata params)
    external
    payable
    ensure(params.deadline)
    nonReentrant
    returns (uint256 amountOut)
{
    ...
    sendFee(feeAmount);
    refundMon(params.to, monUsed, params.amountInMax);

    emit DexRouterBuy(msg.sender, params.token, monUsed, amountOut);
}
```

When `msg.sender` ≠ `params.to`, the caller loses their refund. This is particularly problematic for aggregators or contracts performing swaps on behalf of users.

The same issue exists in `exactOutSell`.

Recommendations:

Refund to `msg.sender`.

Nad.fun: Resolved with `@0a15209842 ...`.

Zenith: Verified.

[M-3] Inconsistent fee calculation leads to under collection

SEVERITY: Medium	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: High

Target

- DexRouter.sol#L98-L100
- DexRouter.sol#L233
- DexRouter.sol#L385

Description:

In DexRouter:

- `buy` correctly calculates fees on the gross amount (`msg.value`), then swaps the net remainder
- `exactOutBuy` and `exactOutSell` incorrectly calculate fees on the net pool amount, effectively charging "fee on net" instead of "fee on gross"

For example: With a 0.3% fee and 100 MON net pool requirement:

- Incorrect (current): Fee = 100 × 3000 / 1M = 0.3 MON → User pays 100.3 MON
- Correct: Gross = 100 × 1M / 997,000 = 100.301 MON → Fee = 0.301 MON

The difference compounds with larger amounts and higher fees.

```
function exactOutBuy(ExactOutBuyParams calldata params)
    external
    payable
    ensure(params.deadline)
    nonReentrant
    returns (uint256 amountOut)
{
    ...
    // 6. Calculate actual wMON used
    uint256 amountIn = uint256(dexData.zeroForOne ? amount0 : amount1);
    uint256 feeAmount = calculateFeeAmount(amountIn);
    uint256 monUsed = amountIn + feeAmount;
    if (monUsed > params.amountInMax) revert InsufficientInput();
    ...
}
```

```
function exactOutSell(ExactOutSellParams calldata params)
    external
    nonReentrant
    ensure(params.deadline)
    returns (uint256 amountOut)
{
    ...
    // 3. Specify exact output (negative): net MON amount user wants to
    receive
    uint256 feeAmount = calculateFeeAmount(params.amountOut); // Fee on net
    uint256 necessaryAmountOut = params.amountOut + feeAmount;

    ...
}
```

```
function calculateFeeAmount(uint256 amount)
    public view returns (uint256 feeAmount) {
    return FullMath.mulDivRoundingUp(amount, routerFee, 1_000_000); // Fee
    on net
}
```

Recommendations:

Calculate fees on gross amounts consistently.:

```
// Correct formula: gross = net × 1M / (1M - r)
uint256 grossAmount = FullMath.mulDivRoundingUp(netAmount, 1_000_000,
    1_000_000 - routerFee);
uint256 feeAmount = grossAmount - netAmount;
```

Nad.fun: Resolved with @7a966597a ... .

Zenith: Verified.

4.3 Low Risk

A total of 11 low risk findings were identified.

[L-1] Unnecessary allowance check against maximum instead of actual transfer amount

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- BondingCurveRouter.sol#L242

Description:

The `exactOutSell` function validates that the user's allowance meets `params.amountInMax`, but only transfers `necessaryAmountIn` tokens (which is typically less than the maximum). This forces users to approve more tokens than actually needed for the transaction.

```
function exactOutSell(ExactOutSellParams calldata params)
    external override ensure(params.deadline) {
    if (params.amountOut <= 0) revert InsufficientAmountOut();
    if (IERC20(params.token).allowance(msg.sender, address(this)) <
    params.amountInMax) {
        revert InvalidAllowance();
    }

    uint256 necessaryAmountIn = getAmountInWithFee(params.token,
    params.amountOut, false);
    if (necessaryAmountIn > params.amountInMax)
    revert InsufficientAmountInMax();

    IERC20(params.token).safeTransferFrom(msg.sender, curve,
    necessaryAmountIn);
    uint256 amountOut = IBondingCurve(curve).sell(address(this),
    params.token); // Transfer the net amount user wants to receive
    uint256 restValue = amountOut - params.amountOut;
    if (restValue > 0) {
        IWMon(wMon).deposit{value:  restValue}();
        IERC20(wMon).safeTransfer(foundationTreasury, restValue);
    }
    TransferHelper.safeTransferMon(params.to, params.amountOut);
}
```

The impact is that users must grant excessive allowances, increasing their risk exposure if the router contract is compromised.

Recommendations:

Validate allowance against the `necessaryAmountIn`.

Nad.fun: Resolved with @cb278020ca ... .

Zenith: Verified.

[L-2] Creator exemption from anti-sniping penalty bypassed in exactOutBuy

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- BondingCurveRouter.sol#L179

Description:

The anti-sniping mechanism exempts token creators from penalties by checking if `to == creator` in `BondingCurve.buy`. However, `exactOutBuy` forwards the buy through the router contract itself, breaking this exemption.

When a creator calls `exactOutBuy`, the function invokes `curve.buy(address(this), token)` with the `router` as the recipient, then forwards tokens to `msg.sender`. Since `to` is the router address (not the creator), the creator check fails and the penalty applies even for legitimate creator purchases.

```
function exactOutBuy(ExactOutBuyParams calldata params)
    external payable override ensure(params.deadline) {
    ...
    uint256 amountOut = IBondingCurve(curve).buy(address(this),
    params.token);

    ...
}
```

```
function buy(address to, address token)
    external
    nonReentrant
    notLocked(token)
    returns (uint256 effectiveAmountOut)
{
    ...
    bool isCreator = ICreatorManager(creatorManager).creators(token) == to;
    ...
}
```

Recommendations:

If `msg.sender` is the creator, route the buy directly to the creator.

Nad.fun: Resolved with [@eb2d5b93de ...](@eb2d5b93de).

Zenith: Verified.

[L-3] exactOutBuy reverts when anti-sniping penalties are active

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- BondingCurveRouter.sol#L174
- BondingCurveRouter.sol#L181

Description:

The `exactOutBuy` function calculates the required input amount using `getAmountInWithFee`, which does not account for anti-sniping penalties. When penalties are active, the actual `amountOut` received from `IBondingCurve(curve).buy` is less than `params.amountOut` due to penalty deductions.

This causes the subtraction `restTokenAmount = amountOut - params.amountOut` to underflow and revert:

```
function exactOutBuy(ExactOutBuyParams calldata params)
    external payable override ensure(params.deadline) {
    if (msg.value ≠ params.amountInMax) revert InsufficientAmountInMax();
    uint256 necessaryAmount = getAmountInWithFee(params.token,
    params.amountOut, true);
    if (necessaryAmount > params.amountInMax)
    revert InsufficientAmountInMax();

    IWMon(wMon).deposit{value:  necessaryAmount}();
    IERC20(wMon).safeTransfer(curve, necessaryAmount);
    uint256 amountOut = IBondingCurve(curve).buy(address(this),
    params.token);

    uint256 restTokenAmount = amountOut - params.amountOut; // Reverts:
    amountOut < params.amountOut
    if (restTokenAmount > 0) {
        IERC20(params.token).safeTransfer(tokenTreasury, restTokenAmount);
    }
     ...
}
```

It means users cannot execute exact output purchases during anti-sniping periods.

Recommendations:

Modify `getAmountInWithFee` to include anti-sniping penalty calculations.

Nad.fun: Resolved with `eb2d5b93de ...` and `269109f230 ...`.

Zenith: Verified.

[L-4] Missing available token check in create function

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- BondingCurveRouter.sol#L94-L102

Description:

The `create` function allows creators to purchase tokens during token creation. However, unlike the `buy` function, it does not validate whether the requested `amountOut` exceeds the available tokens on the bonding curve.

When `params.amountOut` exceeds available supply, `getAmountInWithFee` calculates the input amount for the full requested quantity. The excess portion that cannot be purchased is retained as fees instead of being refunded, resulting in unintended loss for the creator.

```
function create(TokenCreationParams calldata params)
    external
    payable
    override
    returns (address token, address pool)
{
    ...
    (token, pool) = IBondingCurve(curve).create(_params);
    uint256 totalUsed = deployFeeAmount;
    if (params.amountOut > 0) {
        uint256 amountIn = getAmountInWithFee(token, params.amountOut,
    true);
        totalUsed += amountIn;
        if (msg.value < totalUsed) revert InsufficientAmountIn();

        IERC20(wMon).safeTransfer(curve, amountIn);

        IBondingCurve(curve).buy(msg.sender, token);
    }
    //unused fund return
    if (msg.value > totalUsed) {
        uint256 refundAmount = msg.value - totalUsed;
```

```
            IWMon(wMon).withdraw(refundAmount);
            TransferHelper.safeTransferMon(msg.sender, refundAmount);
        }
    }
```

Recommendations:

Add a pre-purchase validation check similar to the `buy` function.

Nad.fun: Resolved with `@808a9b771b ...`.

Zenith: Verified.

[L-5] Event not emitted for non-penalized transactions

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- BondingCurve.sol#L294-L296

Description:

The `buy` function incorrectly determines whether a transaction is penalized, causing the `CurveBuy` event to be suppressed for legitimate, non-penalized purchases.

The function `_calculateFeeAmountWithPenalty` returns the total fee amount (base fee + penalty). When `penalty = 0`, it still returns the base fee amount, which is greater than zero. Consequently, `isPenalized` evaluates to true for all non-creator transactions that include fees, even when no penalty is actually applied.

```
function buy(address to, address token)
    external
    nonReentrant
    notLocked(token)
    returns (uint256 effectiveAmountOut)
{
    ...
    // ========================================
    // 2. Anti-sniping penalty and fee calculation
    // ========================================
    uint256 penaltyAmount = _calculateFeeAmountWithPenalty(token,
    actualAmountIn);
    bool isPenalized = penaltyAmount > 0 && !isCreator; // Always true for
    non-creators with fees
    ...
    // ========================================
    // 8. Emit event (only if not penalized)
    // ========================================
    if (!isPenalized) { // Event never emitted for regular fee-paying users
        emit CurveBuy(to, token, actualAmountIn, effectiveAmountOut);
    }
}
```

This causes `CurveBuy` events to be omitted from legitimate transactions.

Recommendations:

Separate the penalty check from the fee amount. Track whether a penalty was actually applied.

Nad.fun: Resolved with `@9e098526c4 ...`.

Zenith: Verified.

[L-6] Blacklisted actor can still receive fund allocation

SEVERITY: Low	IMPACT: High
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- LpManager.sol#L155

Description:

The protocol only validates actor blacklist status during token creation with a bonding curve. However, if an actor is blacklisted after token creation, `BondingCurve.graduate` can still allocate funds to the blacklisted actor, resulting in potential fund loss.

The `allocate` function retrieves the actor without any blacklist validation:

```
function allocate(AllocateParams calldata params)
    external onlyRole(CURVE_ROLE) {
    PoolData memory poolData = _loadPoolData(params.token);
    poolData.bondingTick = calculateBondingTick(params,
    poolData.monIsToken0, poolData.tickSpacing);
    ILPActor lpActor = ILPActor(getActor(params.token));

    ...

    // Actor deploys liquidity for us and stores the liquidity according to
    the pool address.
    (uint256 totalMonLiquidity, uint256 totalTokenLiquidity)
    = lpActor.mint(poolData, amount0, amount1);

    ...
}
```

Recommendations:

Add a blacklist check in the `allocate` function before retrieving and interacting with the actor.

Nad.fun: Acknowledged. The restriction on create is intentional.

[L-7] Slippage protection bypassed due to estimate-based validation

SEVERITY: Low	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Low

Target

- BondingCurveRouter.sol#L147-L149

Description:

The `BondingCurveRouter.buy` function validates `params.amountOutMin` against an estimated amount (`possibleAmountOut`) rather than the actual tokens received from `IBondingCurve(curve).buy`. This estimate is calculated by `getAmountOutWithFee`, which does not account for anti-sniping penalties applied during execution.

Result: Users may receive fewer tokens than `amountOutMin` without the transaction reverting, effectively bypassing slippage protection.

```
function buy(BuyParams calldata params)
    external payable override ensure(params.deadline) {
    ...

    // Calculate how many tokens can be purchased with the sent MON
    uint256 possibleAmountOut = getAmountOutWithFee(params.token, msg.value,
    true);

    ...

    // ========================================
    // 3. Slippage protection verification
    // ========================================

    // Verify minimum required token amount (slippage protection)
    if (possibleAmountOut < params.amountOutMin) {
        revert InsufficientAmountOut();
    }

    ...
    // ========================================
    // 5. Execute purchase through bonding curve
    // ========================================

    IBondingCurve(curve).buy(params.to, params.token);
}
```

`buy`, `sell`, and `sellPermit` all validate against estimates instead of actual amounts transferred.

Recommendations:

Capture the actual amount returned from the bonding curve and validate it against the slippage parameter:

```
uint256 actualAmountOut = IBondingCurve(curve).buy(params.to, params.token);
if (actualAmountOut < params.amountOutMin) {
    revert InsufficientAmountOut();
}
```

Nad.fun: Resolved with @c197e94d7a ... .

Zenith: Verified.

[L-8] Front-running pool creation causes DOS on token launch

SEVERITY: Low	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- DexDeployer.sol#L50

Description:

The `DexDeployer.createPool` function is vulnerable to a front-running DOS attack. Since tokens are deployed using CREATE2, their addresses are deterministic and can be precomputed. An attacker can front-run the `BondingCurve.create` transaction and deploy a Uniswap V3 pool with the same token/wMon pair and fee tier before the legitimate pool creation occurs.

When `DexDeployer.createPool` subsequently calls `_factory.createPool`, the transaction will revert because the pool already exists, preventing the token from launching successfully.

```
function createPool(address token) external returns (address pool) {
    IUniswapV3Factory _factory = IUniswapV3Factory(factory);
    pool = _factory.createPool(token, wMon, feeTier);

    //INITIALIZE
    (
        uint256 virtualMonReserve,
        uint256 virtualTokenReserve,
        uint256 targetVirtualTokenAmount
    ) = IBondingCurve(curve).config();
    uint256 k = virtualMonReserve * virtualTokenReserve;
    uint256 targetVirtualMonAmount = k / targetVirtualTokenAmount;

    uint160 sqrtPrice = _calculateSqrtPrice(
        targetVirtualMonAmount,
        targetVirtualTokenAmount
    );

    IUniswapV3Pool(pool).initialize(sqrtPrice);
    /// The cardinality required by Ammplify for twap prices.
    IUniswapV3Pool(pool).increaseObservationCardinalityNext(32);

    return pool;
}
```

Recommendations:

Using private mempool to avoid front-running.

Nad.fun: Acknowledged. As discussed, we're assuming that this type of attack wouldn't occur unless it's an intentional frontrun by validators, so we're keeping it acknowledged and documented as such.

[L-9] Residual token allowances not reset after minting

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- AmmplifyFlatActor.sol#L51
- AmmplifyFlatActor.sol#L59

Description:

The `AmmplifyFlatActor.mint` function approves amount0 and amount1 to the Ammplify contract but does not reset unused allowances to zero after minting. Since liquidity positions are minted in halves and may not consume the full approved amount, residual allowances remain. While the function refunds excess tokens to the owner, the approval remains active.

```
function mint(
    ILpManager.PoolData calldata poolData,
    uint256 amount0,
    uint256 amount1
) public override onlyOwner returns (uint256 mint0, uint256 mint1) {
    // Get the amounts first.
    if (amount0 > 0) {
        IERC20(poolData.token0).safeTransferFrom(
            owner,
            address(this),
            amount0
        );
        IERC20(poolData.token0).safeApprove(address(ammplify), amount0);
    }
    if (amount1 > 0) {
        IERC20(poolData.token1).safeTransferFrom(
            owner,
            address(this),
            amount1
        );
        IERC20(poolData.token1).safeApprove(address(ammplify), amount1);
    }
```

```
        ...
        // Refunds tokens but approval remains
        mint0 = amount0 - excessBalance0;
        if (excessBalance0 > 0) {
            IERC20(poolData.token0).safeTransfer(owner, excessBalance0);
        }
        uint256 excessBalance1 = IERC20(poolData.token1).balanceOf(
            address(this)
        );
        mint1 = amount1 - excessBalance1;
        if (excessBalance1 > 0) {
            IERC20(poolData.token1).safeTransfer(owner, excessBalance1);
        }
    }
```

Recommendations:

Reset token allowances to zero after minting operations.

Nad.fun: Resolved with `@024c231a5f ...`.

Zenith: Verified.

[L-10] Reward pool incorrectly accounts for fee-on-transfer tokens

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target:

- RewardPool.sol#L73

Description:

The `addReward` function records the full amount parameter as added rewards, but if the reward token deducts fees during transfer, the actual tokens received will be less. This mismatch causes:

- Rewards to be over-credited in accounting
- Potential insolvency when users attempt to claim full reward amounts

```
function addReward(address token, uint256 amount) external {
    if (token == address(0)) {
        revert InvalidToken();
    }
    if (amount == 0) {
        revert InvalidAmount();
    }
    IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
    uint256 epoch = currentEpoch;
    rewards[token][epoch] += amount;
    emit AddReward(token, msg.sender, amount, epoch);
}
```

Recommendations:

Measure the actual tokens received by comparing balances before and after the transfer.

Nad.fun: Resolved with `@179fee4232a7 ...`

Zenith: Verified.

[L-11] Event emits incorrect token amount in sell function

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- DexRouter.sol#L297

Description:

The `DexRouter.sell` function emits the initial input amount (`params.amountIn`) in the `DexRouterSell` event, rather than the actual amount of tokens sold (`totalAmountIn`).

```
/**
 * @notice Emitted when tokens are sold for MON through the DEX router
 * @param sender Address that initiated the sell transaction
 * @param token Address of the token that was sold
 * @param amountIn Amount of tokens sold
 * @param amountOut Amount of MON tokens received
 */
event DexRouterSell(address indexed sender, address indexed token,
    uint256 amountIn, uint256 amountOut);

function sell(SellParams calldata params)
    external
    nonReentrant
    ensure(params.deadline)
    returns (uint256 amountOut)
{
    ...
    // 6. Tokens actually used (refund excess)
    uint256 totalAmountIn = dexData.zeroForOne ? uint256(amount0) :
    uint256(amount1);
    if (totalAmountIn < params.amountIn) {
        IERC20(params.token).safeTransfer(msg.sender, params.amountIn
    - totalAmountIn);
    }

    ...
    emit DexRouterSell(msg.sender, params.token, params.amountIn,
    amountOut);
}
```

Recommendations:

Emit `totalAmountIn` to accurately reflect the actual tokens sold.

Nad.fun: Resolved with `@8bdffdc02300 ...`

Zenith: Verified.

4.4 Informational

A total of 4 informational findings were identified.

[I-1] Misleading comment in TransferHelper

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- TransferHelper.sol#L17-L19

Description:

The `safeTransferMon` function's documentation incorrectly states it "uses a low-level call with a fixed gas stipend of 2300" that prevents reentrancy. However, the implementation uses `.call{value: amount}("")`, which forwards all remaining gas rather than limiting it to 2300.

```
    * @notice Safely transfers native NATIVE (ETH) to an address
    * @dev Uses a low-level call with a fixed gas stipend of 2300
    * This gas stipend is enough for the receiving contract's fallback
    function,
    * but not enough to make an external call (preventing reentrancy)
    * @param to Address receiving the NATIVE
    * @param amount Amount of NATIVE to transfer in wei
    */
function safeTransferMon(address to, uint256 amount) internal {
    (bool success,) = payable(to).call{value: amount}("");
    require(success, "TransferHelper: Transfer failed");
}
```

Recommendations:

Update the comment to accurately reflect that the function forwards all available gas. Do not rely on `safeTransferMon` for reentrancy protection.

Nad.fun: Acknowledged, will be addressed in a future release.

[I-2] Unreachable overflow check in addActor

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- LpManager.sol#L109

Description:

The `MaxActorsReached` check is unreachable due to the order of operations. Since `lastActorId` is uint8 (max value 255), incrementing it when `lastActorId = 255` causes an overflow revert in Solidity 0.8+. The subsequent check `if (lastActorId ≥ 256)` never execute.

```
function addActor(address actor) external onlyRole(DEFAULT_ADMIN_ROLE)
    returns (uint8) {
    lastActorId++;
    if (lastActorId >= 256) revert MaxActorsReached();
    actors[lastActorId] = actor;
    emit AddActor(lastActorId, actor);
    return lastActorId;
}
```

Recommendations:

Perform the boundary check before incrementing:

```
function addActor(address actor) external onlyRole(DEFAULT_ADMIN_ROLE)
    returns (uint8) {
    if (lastActorId >= 255) revert MaxActorsReached();
    lastActorId++;
    actors[lastActorId] = actor;
    emit AddActor(lastActorId, actor);
    return lastActorId;
}
```

Nad.fun: Resolved with `@8aea2648a8c ...`

Zenith: Verified.

[I-3] Missing allowance validation in sellPermit

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- BondingCurveRouter.sol#L222-L224

Description:

The `sellPermit` function lacks allowance validation that exists in the `sell` function. The permit grants `params.amountAllowance` approval, but the actual transfer uses `params.amountIn`. They can be different.

```
function sellPermit(SellPermitParams calldata params)
    external override ensure(params.deadline) {
    if (params.amountIn <= 0) revert InsufficientAmountIn();
    IERC20Permit(params.token).permit(
            msg.sender, address(this), params.amountAllowance,
    params.deadline, params.v, params.r, params.s
    );

    // Get the user's net amount after fees
    uint256 netAmountOut = getAmountOutWithFee(params.token,
    params.amountIn, false);
    if (netAmountOut < params.amountOutMin) revert InsufficientAmountOut();

    IERC20(params.token).safeTransferFrom(msg.sender, curve,
    params.amountIn);
    IBondingCurve(curve).sell(params.to, params.token); // Now passes the
    net amount user will receive
}
```

Recommendations:

Add allowance validation for consistency with the `sell` function.

Nad.fun: Acknowledged.

[I-4] Unused ReentrancyGuard inheritance

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- BondingCurveRouter.sol#L21

Description:

The `BondingCurveRouter` contract inherits `ReentrancyGuard` but never applies the `nonReentrant` modifier to any functions.

```
contract BondingCurveRouter is IBondingCurveRouter, ReentrancyGuard {
```

Recommendations:

Either:

- Apply the `nonReentrant` modifier to external functions that interact with external contracts or handle value transfers, or
- Remove the `ReentrancyGuard` inheritance.

Nad.fun: Resolved with `@c197e94d7a ...`.

Zenith: Verified.