SHERLOCK

Security Review For native

Private Bug Bounty Audit Contest Prepared For:	**native**
Lead Security Expert:	**hildingr**
Date Audited:	**May 23 - May 29, 2025**
Final Commit:	**6745b1d**

Introduction

Native is an on-chain platform to build token liquidity that is openly accessible and cost effective. It serves as an alternative to traditional AMMs through integration of two innovative designs: the Native Swap Engine and Native Credit Pool. The contest's focus is on ensuring fund safety in the credit pool and the usage of the fund, through Swap Engine.

Scope

Repository: Native-org/v2-core

Audited Commit: 6ee5ef69f8a7f2e0435e1fc65dc3f34786a177f7

Final Commit: 6745b1deb50eda266ebcc4d724cff0c79448df83

Files:

- src/CreditVault.sol
- src/NativeLPToken.sol
- src/NativeRFQPool.sol
- src/NativeRouter.sol
- src/interfaces/IQuote.sol
- src/libraries/ConstantsLib.sol
- src/libraries/ExternalSwap.sol
- src/libraries/ReentrancyGuardTransient.sol

Final Commit Hash

6745b1deb50eda266ebcc4d724cff0c79448df83

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
3	11

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0xAadi
0xShoonya
0xSolus
0xaxaxa
0xc0ffEE
0xrex
John44
Kirkeelee
Kose
aslanbek
dobrevaleri
eeyore
hildingr
iamandreiski
ifeco445
jasonxiale
montecristo
moray5554
newspacexyz
tobi0x18

Issue H-1: Users will be charged 2x fees and during multiHop swaps will lose tokens to the router

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/92

Found by

0xShoonya, 0xSolus, 0xrex, John44, Kirkeelee, eeyore, montecristo, newspacexyz

Summary

There is a current deduction of fee charged to the user from the amount being traded which is excessive and shouldn't be deducted since each user already pays for the trading fee of the trade out of pocket. As a result of this, a lot of issues spawn up that ultimately leads to the user losing tokens overall either to the router or more charged fees and even less output tokens received.

Root Cause

```
function tradeRFQT(
        RFQTQuote memory quote,
        uint256 actualSellerAmount,
        uint256 actualMinOutputAmount
    ) external payable override nonReentrant whenNotPaused {
        ...

        // cut widget fee based on the actual amount
@>        effectiveSellerTokenAmount =
            _transferSellerToken(quote.multiHop, payee, quote.sellerToken,
↪   effectiveSellerTokenAmount, quote.widgetFee);

        if (isNativePool) {
@>
↪   NativeRFQPool(payable(quote.pool)).tradeRFQT(effectiveSellerTokenAmount, quote);
        } else if (whitelistRouter[quote.pool]) {
            ...
    }
```

```
function _transferSellerToken(
        bool multiHop,
        address payee,
        address sellerToken,
```

```
        uint256 sellerTokenAmount,
        WidgetFee memory widgetFee
    ) internal returns (uint256 effectiveSellerTokenAmount) {
        if (msg.value > 0 && !multiHop) {
            ...
        } else {
@>            effectiveSellerTokenAmount = _chargeWidgetFee(widgetFee,
↪   sellerTokenAmount, sellerToken, false);

            if (multiHop) {
@>                TransferHelper.safeTransfer(sellerToken, payee,
↪   effectiveSellerTokenAmount);
            } else {
@>                TransferHelper.safeTransferFrom(sellerToken, msg.sender, payee,
↪   effectiveSellerTokenAmount);
            }
        }
    }
```

```
function _chargeWidgetFee(
        WidgetFee memory widgetFee,
        uint256 amountIn,
        address sellerToken,
        bool hasAlreadyPaid
    ) internal returns (uint256) {
        uint256 fee = widgetFee.feeRate > 0 ? (amountIn * widgetFee.feeRate) /
↪   10_000 : 0;

        if (fee > 0) {
            TransferHelper.safeTransferFrom(
                sellerToken, hasAlreadyPaid ? address(this) : msg.sender,
↪   widgetFee.feeRecipient, fee
            );
            emit WidgetFeeTransfer(widgetFee.feeRecipient, widgetFee.feeRate, fee,
↪   sellerToken);

@>            amountIn -= fee;
        }

@>        return amountIn;
    }
```

A little primer on what the protocol intends to do:

1. Alice tries to trade 10 WETH for USDC when 1 WETH is currently worth 3000 USDC
2. The `widgetFee.feeRate` to facilitate this trade is 1% hence Alice should receive 30,000 USDC while 300 USDC will be taken from Alice separately using `safeTransferFrom` inside the `_chargeWidgetFee` function as can be seen below:

```
if (fee > 0) {
            TransferHelper.safeTransferFrom(
                sellerToken, hasAlreadyPaid ? address(this) : msg.sender,
↪  widgetFee.feeRecipient, fee
            );
```

3. However, this is not the current case and Alice will be double charged the 1% fee.

This happens because inside the `_chargeWidgetFee` function, we first take 1% fee from Alice using `safeTransferFrom` then we go on again to also deduct 1% from the amount Alice intends to swap. The correct implementation is to deduct once either do it by the `safeTransferFrom` or do it by deducting the 1% from the swap about by doing `amountIn -= fee` but don't do both.

4. Because of this deductions and transfers Alice will be double charged and also if Alice does a multiHop swap, some tokens will be left in the contract corresponding to the 1% fee for the swap which will be locked and should have been swapped during Alice's trade.

The `_chargeWidgetFee` returns an amountIn that influences what the `effectiveSellerTokenAmount` that is used in multiple parts of the internal functions and outer functions that the `tradeRFQT` function of the NativeRouter contract calls. Since this amount is already flawed, all of the instances where it is used will be flawed as well the external swap logic where the `sellerTokenAmount: quote.sellerTokenAmount`, still remains the initial amount the market maker signed for the qoute without the fee accounted. For the cases of external router, swaps the `sellerTokenAmount: quote.sellerTokenAmount`, will already be higher than the `effectiveSellerTokenAmount` which will also cause the external swap to fail as we later approve that amount in the externalSwap function call even though the NativeRouter contract has less amount than the `quote.sellerTokenAmount` states here `sellerTokenAmount: quote.sellerTokenAmount,`.

https://github.com/sherlock-audit/2025-05-native-smart-contract-v2/blob/main/v2-core/src/NativeRouter.sol#L112-L116 https://github.com/sherlock-audit/2025-05-native-smart-contract-v2/blob/main/v2-core/src/NativeRouter.sol#L263-L270 https://github.com/sherlock-audit/2025-05-native-smart-contract-v2/blob/main/v2-core/src/NativeRouter.sol#L281-L290

Internal Pre-conditions

There are no internal conditions for this to occur. The fee deductions and transfers logic in the current code implementation is just flawed.

External Pre-conditions

Assume there is a fee set for the `widgetFee.feeRate` such as 0.1%, 0.01% 0.5% any amount of fee basis point at all.

Attack Path

N/A. Once, there is a `widgetFee.feeRate` set for facilitating trading, the issue comes to fruition for all users of that trading pairs.

Impact

As a result of the over deductions of fees whereby the same fee has already been pulled from the user, the user will be:

1. Double charged in fees
2. In multiHop trades, some portion of the second token trade will be locked in the NativeRouter contract
3. The user will end up receiving less tokens than bargained for in the ending token of a multiHop swap as well as a normal swap

PoC

In the NativeRouter.t.sol test file, make this diff change inside `_createQuote` to begin the setup for a multiHop trade

```
function _createQuote(
        address poolAddress,
        uint256 sellAmount,
        uint256 expectedUsdcAmount
    ) internal view returns (IQuote.RFQTQuote memory) {
        IQuote.RFQTQuote memory quote = IQuote.RFQTQuote({
            pool: poolAddress,
            signer: halo,
-           recipient: alice,
+           recipient: address(router),
            sellerToken: address(weth),
            buyerToken: address(USDC),
            sellerTokenAmount: sellAmount,
            buyerTokenAmount: expectedUsdcAmount,
            amountOutMinimum: expectedUsdcAmount,
            deadlineTimestamp: block.timestamp + 1 hours,
            nonce: 0,
            multiHop: false,
            signature: bytes(""),
            externalSwapCalldata: bytes(""),
            quoteId: bytes16(0),
-            widgetFee: IQuote.WidgetFee({feeRecipient: deployer, feeRate: 0}),
+            widgetFee: IQuote.WidgetFee({feeRecipient: address(1), feeRate: 100}),
            widgetFeeSignature: bytes("")
        });
```

```
        // Generate quote signature
        bytes32 quoteDigest = SignatureUtils.generateQuoteSignature(quote);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(haloKey, quoteDigest);
        quote.signature = abi.encodePacked(r, s, v);

        // Generate widget fee signature
        bytes32 widgetFeeDigest = SignatureUtils.generateWidgetFeeSignature(quote,
↪  alice, address(router));
        (v, r, s) = vm.sign(deployerKey, widgetFeeDigest);
        quote.widgetFeeSignature = abi.encodePacked(r, s, v);

        return quote;
    }
```

Add this function below inside the `NativeRouter.t.sol` file to facilitate the market maker creating the second hop trade in a multiHop trade

```
function _createQuoteTokka(
        address poolAddress,
        uint256 sellAmount,
        uint256 expectedWbtcAmount
    ) internal view returns (IQuote.RFQTQuote memory) {
        IQuote.RFQTQuote memory quote = IQuote.RFQTQuote({
            pool: poolAddress,
            signer: tokka,
            recipient: alice,
            sellerToken: address(USDC),
            buyerToken: address(wbtc),
            sellerTokenAmount: sellAmount,
            buyerTokenAmount: expectedWbtcAmount,
            amountOutMinimum: expectedWbtcAmount,
            deadlineTimestamp: block.timestamp + 1 hours,
            nonce: 1,
            multiHop: true,
            signature: bytes(""),
            externalSwapCalldata: bytes(""),
            quoteId: bytes16(0),
            widgetFee: IQuote.WidgetFee({feeRecipient: address(1), feeRate: 100}),
            widgetFeeSignature: bytes("")
        });

        // Generate quote signature
        bytes32 quoteDigest = SignatureUtils.generateQuoteSignature(quote);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(tokkaKey, quoteDigest);
        quote.signature = abi.encodePacked(r, s, v);

        // Generate widget fee signature
        bytes32 widgetFeeDigest = SignatureUtils.generateWidgetFeeSignature(quote,
↪  alice, address(router));
```

```
        (v, r, s) = vm.sign(deployerKey, widgetFeeDigest);
        quote.widgetFeeSignature = abi.encodePacked(r, s, v);

        return quote;
    }
```

Then paste this PoC in the NativeRouter.t.sol file and run with test with `forge test --mt test_doubeChargesAndLostTokensPoC -vvv`

```
function test_doubeChargesAndLostTokensPoC() public {
        setupPMM();

        // Mint WETH for Alice
        deal(address(weth), alice, 10 ether);

        // deal trade tokens to the tokka treasury
        deal(address(wbtc), tokka, 1_000_000e8);

        // Store initial balances first
        uint256 aliceInitialEth = weth.balanceOf(alice);
        uint256 aliceInitialUsdc = USDC.balanceOf(alice);
        uint256 routerInitialEth = weth.balanceOf(address(router));
        uint256 routerInitialUSDC = USDC.balanceOf(address(router));

        console.log("Alice initialETH: ", aliceInitialEth);
        console.log("Alice initialUSDC: ", aliceInitialUsdc);
        console.log("Router initialETH: ", routerInitialEth);
        console.log("Router initialUSDC: ", routerInitialUSDC);

        // Calculate amounts
        uint256 ethPriceInUsdc = 3000 * 10 ** 6;
        uint256 sellAmount = 10e18; // selling WETH
        uint256 expectedUsdcAmount = sellAmount * ethPriceInUsdc / 1 ether; //
↪  expects 30k USDC

        vm.prank(halo);
        USDC.approve(address(haloRFQPool), type(uint256).max);
        vm.prank(tokka);
        wbtc.approve(address(tokkaRFQPool), type(uint256).max);

        // Create and execute the first hop trade
        IQuote.RFQTQuote memory quote = _createQuote(address(haloRFQPool),
↪  sellAmount, expectedUsdcAmount);

        vm.startPrank(alice);
        weth.approve(address(router), type(uint256).max);
        router.tradeRFQT(quote, quote.sellerTokenAmount, quote.amountOutMinimum);
        vm.stopPrank();
```

```
        console.log("Alice endingETH: ", weth.balanceOf(alice));
        console.log("Alice endingUSDC: ", USDC.balanceOf(alice));
        console.log("Router endingETH: ", weth.balanceOf(address(router)));
        console.log("Router endingUSDC: ", USDC.balanceOf(address(router)));

        // now do the second hop
        uint256 wbtcPriceInUsdc = 90000 * 10 ** 6;
        uint256 sellAmount1 = 29700 * 10 ** 6; // selling 27k USDC
        uint256 expectedWbtcAmount = sellAmount1 * (10 ** 8) / wbtcPriceInUsdc;
        console.log("expected WBTC amount: ", expectedWbtcAmount);

        // Create and execute the second hop trade
        IQuote.RFQTQuote memory quote2 = _createQuoteTokka(address(tokkaRFQPool),
↪   sellAmount1, expectedWbtcAmount);
        vm.startPrank(alice);
        USDC.approve(address(router), type(uint256).max);
        router.tradeRFQT(quote2, quote2.sellerTokenAmount, quote2.amountOutMinimum);
        vm.stopPrank();

        console.log("Alice endingETH: ", weth.balanceOf(alice));
        console.log("Alice endingUSDC: ", USDC.balanceOf(alice));
        console.log("Alice endingWbtc: ", wbtc.balanceOf(alice));
        console.log("Router endingETH: ", weth.balanceOf(address(router)));
        console.log("Router endingUSDC: ", USDC.balanceOf(address(router)));
        console.log("Router endingWbtc: ", wbtc.balanceOf(address(router)));
    }
```

After running the above test, we will notice the following issues:

1. The user is over charged fees
2. The user also loses some USDC that is now locked in the Router contract
3. The user receives less WBTC than they should

```
 [PASS] test_doubeChargesAndLostTokensPoC() (gas: 12570730)
Logs:
  Alice initialETH:  10000000000000000000
  Alice initialUSDC:  1000000000000
  Router initialETH:  0
  Router initialUSDC:  0
  Alice endingETH:  0
  Alice endingUSDC:  1000000000000
  Router endingETH:  0
  Router endingUSDC:  29700000000
  expected WBTC amount:  33000000
  Alice endingETH:  0
  Alice endingUSDC:  999703000000
  Alice endingWbtc:  32670000
  Router endingETH:  0
  Router endingUSDC:  297000000
```

```
  Router endingWbtc:  0
```

Mitigation

The mitigation below is a possible fix that resolves all the issues described in this report:

```
function _chargeWidgetFee(
        WidgetFee memory widgetFee,
        uint256 amountIn,
        address sellerToken,
        bool hasAlreadyPaid
    ) internal returns (uint256) {
        uint256 fee = widgetFee.feeRate > 0 ? (amountIn * widgetFee.feeRate) /
↪  10_000 : 0;

        if (fee > 0) {
            TransferHelper.safeTransferFrom(
                sellerToken, hasAlreadyPaid ? address(this) : msg.sender,
↪  widgetFee.feeRecipient, fee
            );
            emit WidgetFeeTransfer(widgetFee.feeRecipient, widgetFee.feeRate, fee,
↪  sellerToken);

-            amountIn -= fee;
        }

        return amountIn;
    }
```

Do the above fix or transfer the fee from the user inside if condition below and keep the `amountIn` the same without deducting `fee` from it. Either one is okay, just don't transfer fees and still deduct the same fee from `amountIn`.

```
if (fee > 0) {
            TransferHelper.safeTransferFrom(
                sellerToken, hasAlreadyPaid ? address(this) : msg.sender,
↪  widgetFee.feeRecipient, fee
);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Native-org/v2-core/commit/5a744ec472bb93badd0306e38d89aca017343f54

Issue H-2: Exploit of yield mechanism

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/143

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

hildingr

Summary

The yield distribution mechanism can be exploited by front-running epoch-updates and withdrawing after two yield distributions. An attacker can double their yield-to-capital-commitment ratio by exploiting the yield distribution mechanism when the default values are used for `minRedeemInterval` and `EPOCH_UPDATE_INTERVAL`. All other LPs yield is diluted since the attacker receives more than expected.

Root Cause

Yield is distributed to all LP holders regardless of deposit time, this allow attackers to front-run the distribution and redeem between distribution events.

Yield is distributed every 8h when `epochUpdate()` is called *

```
lpTokens[token].distributeYield(fundingFee);
```

A user can deposit right before this and receive the yield. The user can the redeem after `minRedeemInterval` has passed and then enter again before `epochUpdate()` is called.

When the user redeems after `minRedeemInterval` has passed no fee is paid.*

```
if (
    block.timestamp < lastDepositTimestamp[msg.sender] + minRedeemInterval &&
↪   earlyWithdrawFeeBips > 0
        && !redeemCooldownExempt[msg.sender]
) {
    underlyingAmount -= (underlyingAmount * earlyWithdrawFeeBips) / 10_000;
}
```

Internal Pre-conditions

1. Funding fees need to be accrued in the system to be distributed as yield

External Pre-conditions

None

Attack Path

1. **Attacker deposits funds into the LP pool just before the `epochUpdate` function is called.**
2. **The `epochUpdate` function is called and distributes yield to all current LP token holders, including the attacker who just deposited**
3. **Attacker waits in the pool for another ~8> hours until a second `epochUpdate` occurs**
4. **The second `epochUpdate` distributes yield again to all LP token holders, including the attacker**
5. **Attacker redeems their funds immediately after receiving the second yield distribution without paying a fee since more than 8 has passed**
6. **Attacker repeats this cycle every ~16 hours**

Impact

The existing LP token holders suffer a dilution of yield as the attacker receives a disproportionate amount of yield compared to their time commitment. The attacker effectively doubles their yield-to-capital-commitment ratio by only staying in the pool for 8 hours but receiving yield for 16 hours worth of capital commitment.

PoC

No response

Mitigation

No response

Discussion

Ethronaut

Hi, this is not an issue, we discussed it internally a long time ago, if a front-running LP holder is willing to keep their funds in our credit vault for **more than 8 hours**, they're welcome to do so. honestly, I've never seen a front-running hacker with that much patience

WangSecurity

Although it may be true that most front-running attackers are not "with that much patience", the attack is quite likely, since no funds are put at risk.

So it's not bothersome for the attacker to use this attack pattern. There are ways through which, even if the utilization is high, the attacker can manage to withdraw his funds (by performing a swap, for instance).

The idea is that, for instance, if normal users get 2x yield in 2 epoch distributions for their commitment, the attacker gets 2x distributions worth of yield in just `1 epoch distribution + 1 block`, without putting any funds at risk.

Ethronaut

@WangSecurity, understood, as we acknowledged when designing the `earlyWithdrawFee` feature, if a user is willing to keep their funds locked in our credit vault for more than 8 hours, we're willing to accommodate that.

Please include this response in the final audit report, thanks

Issue H-3: Inflation Attack possible through redeem function

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/144

Found by

0xaxaxa, John44, aslanbek, eeyore, hildingr

Summary

An inflation attack is possible since the penalty taken in the `redeem()` stays in the vault and inflates the share price.

Root Cause

When shares are redeemed a fee is taken *

```
if (
    block.timestamp < lastDepositTimestamp[msg.sender] + minRedeemInterval &&
↪   earlyWithdrawFeeBips > 0
        && !redeemCooldownExempt[msg.sender]
) {
    underlyingAmount -= (underlyingAmount * earlyWithdrawFeeBips) / 10_000;
}
```

The fee amount remains in the `CreditVault` and is still accounted in the `underlyingAmount` for the remaining shares.

Redeeming early can inflates the share price since the the total amount `sharesToBurn` is burned while the corresponding `underlyingAmount` is not removed since the fee remains.

When a pool has been inflated the subsequent depositors will receive less shares then expected due to rounding down in share calculation *

```
if (totalShares == 0) {
    sharesToMint = amount; // Initial shares 1:1
} else {
    sharesToMint = (amount * totalShares) / totalUnderlying;
}
```

Internal Pre-conditions

1. EarlyWithdrawFeeBips !=0

This inflation attack happens as the pool is newly created. Most likely it will be done by front-running a large deposit to steal a substantial portion.

External Pre-conditions

None

Attack Path

Assume large deposit of 12000 USDC is in the mempool. The attacker will front-run the user and deposit and then redeem all but 1 share to poison the pool

1. Attacker deposit 600001 and receives 600001 shares
2. Attacker redeems 600000 shares. 1% penalty is taken such that `totalUnderlying = 6000 + 1 = 6001 USDC`. Attacker now holds 1 share with 6001 USDC `totalUnderlying`.
3. User deposit is now executed. User `sharesToMint  = 12000*1/6001 =1` since 1.99 is rounded down to 1.

total underlying is now 12000 + 6100 = 18001. Attacker holds 50% of the shares and can redeem 9000 USDC and steal 3000 USDC from the user.

Impact

User lose a substantial amount off funds as outlined in the attack paths above.

PoC

No response

Mitigation

Give `earlyWithdrawFee` to a fee recipient so that it does not affect the exchange rate

It is also a good practice to burn a small amount in a deposit in the constructor such that an attacker never can own all the shares.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Native-org/v2-core/commit/3280bb66fb8dce6657912759fd03419e603ff9f1

Issue M-1: Users can avoid earlyWithdrawFee with small sharesToBurn

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/25

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

aslanbek, newspacexyz

Summary

In `_redeem` function of `NativeLPToken` contract, there is no min amount check and users can avoid `earlyWithdrawFee` by splitting share into small amount.

Root Cause

Looking into _redeem function of `NativeLPToken` contract, you can notice that users can avoid `earlyWithdrawFee` with small `sharesToBurn` :

```
    function _redeem(uint256 sharesToBurn, address to) internal returns (uint256
↪   underlyingAmount) {
        require(sharesToBurn > 0, ErrorsLib.ZeroAmount());
        require(shares[msg.sender] >= sharesToBurn, ErrorsLib.InsufficientShares());

        // Calculate underlying amount
        underlyingAmount = (sharesToBurn * totalUnderlying) / totalShares;

        if (
            block.timestamp < lastDepositTimestamp[msg.sender] + minRedeemInterval
↪   && earlyWithdrawFeeBips > 0
                && !redeemCooldownExempt[msg.sender]
        ) {
@>          underlyingAmount -= (underlyingAmount * earlyWithdrawFeeBips) / 10_000;
        }

        ...
    }
```

Let's imagine that `earlyWithdrawFeeBips` is 1%(100 in bps).
In this case, if `underlyingAmount` is 99 wei, `earlyWithdrawFee` is 0.
If user repeats this operation, there is no loss due to early withdraw.

Considering the gas price, with WBTC(8 decimal and over 100000 usd), 1 wei is worth than gas price on some L2.
This means that users can get profit and also protocol gets loss of no early withdraw fee.

Internal Pre-conditions

NativeLPToken with WBTC

External Pre-conditions

L2 with low gas price

Attack Path

.

Impact

Loss of protocol from no early withdraw fee

Mitigation

Introduce min threshold in `_redeem` function or use rounding up with early withdraw fee calculation.

Discussion

Ethronaut

Hi, this is not an issue, a **very small shareToBurn** amount has **no economic sense**. if someone tries to avoid the early withdrawal fee this way, t**he frontrunning profit from such a tiny token amount would be zero**, and it wouldn't cover the **gas costs**.

WangSecurity

It was computed that on certain chains (e.g. Arbitrum and Mantle), with high-value tokens such as, specifically, **WBTC**, the gas fees are 3x lower than the potential ”early withdrawal” fee penalty.

Thus, technically, on these chains it is profitable for the attacker; also, the incentive to perform it may be to get the same return `underlyingAmount`-wise (actually, it was computed that ~70% of the penalty can be avoided) but also deprive the LPs of the otherwise redistributed penalty fee.

Ethronaut

As you suggested, in an extreme case, the only scenario where profit might be possible is if the WBTC NativeLP token is deployed on Arbitrum.

actually we will apply different early withdrawal fee rates depending on the value and decimals of the underlying token, 1% will not be a realistic fee rate for our NativeLP token, currently the contract have maximum fee cap of 10%.

WangSecurity

> As you suggested, in an extreme case, the only scenario where profit might be possible is if the WBTC NativeLP token is deployed on Arbitrum.

But that still means the scenario is possible, and based on our guidelines, it's considered an extensive constraint, which makes the vulnerability a Medium severity.

> actually we will apply different early withdrawal fee rates depending on the value and decimals of the underlying token, 1% will not be a realistic fee rate for our NativeLP token, currently the contract have maximum fee cap of 10%.

We didn't know how the early withdrawal fee value would be decided, and a 1% fee, considering the max is 10%, seems reasonable.

That's why the decision was to validate an issue.

Ethronaut

@WangSecurity from a purely mathematical point of view, it is possible, but, we've decided not to fix this extremely edge case.

If you take a look at our epochUpdate function, the **maximum yield distribution per update is capped at 1%**, and each epoch update occurs **every 8 hours**. This translates to an annualized simple interest rate of 1000%, which is clearly unrealistic.

Let's consider a more reasonable scenario: assuming an 8-hour funding rate that results in a token yield of 0.1% per epoch update (which is still quite high and equates to ~100% APY). In this case, a user would have to deposit 1,000 wei WBTC just to earn 1 wei of yield.

To avoid being charged the early withdrawal fee through front-running, **if the earlyWithdrawFeeBips is just 1%**, a user can only redeem 99 wei per attempt. This means they would need to call the function 11 times to fully redeem

If the fee is 10%, they would only be able to redeem 9 wei per attempt – requiring 112 calls in total

We believe this behavior is manageable and controlled, and therefore we won't be fixing this issue.

Please kindly include this response in the final audit report, thanks

Issue M-2: quote.nonce is not checked and updated in NativeRouter.tradeRFQT

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/41

Found by

0xSolus, 0xc0ffEE, John44, dobrevaleri, iamandreiski, ifeco445, jasonxiale, newspacexyz

Summary

`quote.nonce` is not checked and updated in `NativeRouter.tradeRFQT`

Root Cause

While `NativeRouter.tradeRFQT` is called, the signature will be verified in _verifyRFQSignature, but the issue is `quote.nonce` **is not updated during the call, which means the same signature can be reused if** `quote.pool` **is external router**

```
 77     function tradeRFQT(
 78         RFQTQuote memory quote,
 79         uint256 actualSellerAmount,
 80         uint256 actualMinOutputAmount
 81     ) external payable override nonReentrant whenNotPaused {
 82         require(quote.widgetFee.feeRate <= MAX_WIDGET_FEE_BIPS,
↪  ErrorsLib.InvalidWidgetFeeRate());
 83         require(block.timestamp <= quote.deadlineTimestamp,
↪  ErrorsLib.QuoteExpired());
 84
 85         _verifyRFQSignature(quote);
 86
     ...
115         if (isNativePool) {
116
↪  NativeRFQPool(payable(quote.pool)).tradeRFQT(effectiveSellerTokenAmount, quote);
117         } else if (whitelistRouter[quote.pool]) {
>>> because the `quote.nonce` is not checked and updated, this branch can be called
↪  multiple times
118             Orders.Order memory order = Orders.Order({
119                 id: 0, // not used
120                 signer: address(0), // not used
121                 buyer: quote.pool,
122                 seller: address(0), // not used
123                 buyerToken: quote.buyerToken,
```

```
124                 sellerToken: quote.sellerToken,
125                 buyerTokenAmount: quote.buyerTokenAmount,
126                 sellerTokenAmount: quote.sellerTokenAmount,
127                 deadlineTimestamp: quote.deadlineTimestamp,
128                 caller: msg.sender,
129                 quoteId: quote.quoteId
130             });
131
132             uint256 actualAmountOut = ExternalSwap.externalSwap(
133                 order, effectiveSellerTokenAmount, quote.recipient,
    ↪ address(this), quote.externalSwapCalldata
134             );
135
136             require(
137                 actualAmountOut >= quote.amountOutMinimum,
138                 ErrorsLib.NotEnoughAmountOut(actualAmountOut,
    ↪ quote.amountOutMinimum)
139             );
140         } else {
141             revert ErrorsLib.InvalidNativePool();
142         }
143     }
```

Internal Pre-conditions

None

External Pre-conditions

external pool is used for swap tokens

Attack Path

swap using external pool

Impact

1. break the invariants listed in readme

> Each nonce must be used exactly once

2. the user can use the same signature to call `NativeRouter.tradeRFQT` mulitple time, which might damage the protocol

PoC

No response

Mitigation

No response

Discussion

Ethronaut

Let me explain the logic of the `tradeRFQT` function. We support two trading modes:

`RFQ Trades` – **where the swapper trades directly with our partnered market makers.**

`External Swaps` – **where the swap is executed through external, whitelisted router contracts** (most of which use an AMM mechanism), **such as the aggregator contracts of 1inch or OpenOcean.**

The key difference between these two modes – lies in **who the counterparty is:**

If the counterparty is our partnered market maker, we must use a `nonce` to prevent replay attacks, since the quote has a time limit (e.g., 60 seconds).

If the counterparty is a **whitelisted** aggregator contract, the quote is real-time and executed on-chain, so there's no need for a nonce, **the nonce field will be default 0**

Additionally, in the externalSwap logic, we already verify that the output amount is greater than or equal to the expected `buyerTokenAmount`.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Native-org/v2-core/commit/6745b1deb50eda266ebcc4d724cff0c79448df83

Issue M-3: _calculateTokenAmount doesn't adjust the buyer amount when the effective amount is larger than the initial seller amount

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/45

Found by

0xaxaxa, 0xc0ffEE, 0xrex, John44, dobrevaleri, iamandreiski, newspacexyz, tobi0x18

Summary

When a native router transaction is initiated, the initiator can change the effective seller amount up to 10%, either below or above the amount outlined in the initial RFQT order. The problem is that within the ExternalSwap library, and more specifically the `_calculateTokenAmount`, the buyerAmount isn't adjusted to match the adjusted seller amount, unlike in the native pool where this is done for both scenarios.

Root Cause

Whenever the effective amount passed to the ExternalSwap library is larger than the initial seller amount outlined in the order, the buyerAmount won't be modified:

```
function _calculateTokenAmount(
       uint256 flexibleAmount,
       Orders.Order memory _order
   ) internal pure returns (uint256, uint256) {
       uint256 buyerTokenAmount = _order.buyerTokenAmount;
       uint256 sellerTokenAmount = _order.sellerTokenAmount;

       require(sellerTokenAmount > 0 && buyerTokenAmount > 0 && flexibleAmount > 0,
↪  ErrorsLib.ZeroAmount());
       if (flexibleAmount < sellerTokenAmount) {
           buyerTokenAmount = FullMath.mulDiv(flexibleAmount, buyerTokenAmount,
↪  sellerTokenAmount);
           sellerTokenAmount = flexibleAmount;
       }
       require(buyerTokenAmount > 0, ErrorsLib.ZeroAmount());
       return (buyerTokenAmount, sellerTokenAmount);
   }
```

As a reference, this is how the case is handled within the Native RFQ Pool where both the < and > cases are handled:

```
_buyerTokenAmount = effectiveSellerTokenAmount != sellerTokenAmount

      ? (effectiveSellerTokenAmount * buyerTokenAmount) / sellerTokenAmount
      : buyerTokenAmount;
```

This is a problem as whenever the seller decides to increase the seller amount, they won't receive a proportionally increased buyer amount which would lead to two possible outcomes.

- These trades frequently fail and the effective amount can't be increased (core functionality is broken) due to slippage set by the original initiator of the RFQ transaction:

```
uint256 actualAmountOut = ExternalSwap.externalSwap(
                order, effectiveSellerTokenAmount, quote.recipient, address(this),
↪   quote.externalSwapCalldata
            );

            require(
                actualAmountOut >= quote.amountOutMinimum,
                ErrorsLib.NotEnoughAmountOut(actualAmountOut,
↪   quote.amountOutMinimum)
            );
```

Or if one isn't set, this would never be caught by the "automated" slippage checks within externalSwap, as the buyerAmount was never modified:

```
 SwapState memory state;
        (state.buyerTokenAmount, state.sellerTokenAmount) =
↪   _calculateTokenAmount(flexibleAmount, order);

require(amountOut >= state.buyerTokenAmount, ErrorsLib.NotEnoughTokenReceived());
```

Internal Pre-conditions

N/A

External Pre-conditions

1. User inputs an effectiveAmount greater than the order seller amount (within the 10% range);
2. The RFQ creator utilizes a non-native pool and the transaction is performed via an ExternalSwap.

Attack Path

1. User initiates a RFQT transaction with an effective amount larger than the sellerTokenAmount, and a non-native pool so that an ExternalSwap is utilized.
2. The contract logic never adjusts the buyer amount to match the modified sellerTokenAmount.
3. The transaction either fails or goes through at the expense of the user since the buyerAmount was never modified, the automated slippage checks didn't catch it, and the user is automatically negatively affected.

Impact

The buyerTokenAmount isn't adjusted whenever the effective amount is larger than the sellerTokenAmount leading to either failed RFQT transactions or users receiving less buyerToken than they're entitled to.

PoC

No response

Mitigation

Adjust the buyerTokenAmount whenever the effective amount is both less than or more than the sellerTokenAmount (i.e. use != as it is in the native RFQ pool).

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Native-org/v2-core/commit/3c8930be86f62501dd5079795bf20b8041133750

Issue M-4: Native won’t be able to Operate on Mantle Chain Because of Transient Storage Usage

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/46

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

Kose

Summary

Mantle Chain does not support transient storage currently. However all entry functions for Native utilizes `nonReentrant` modifier from ReentrancyGuardTransient.sol. Hence all these functions will revert.

Root Cause

Native states that protocol will be deployed in Mantle Chain in README. However with its current implementation, Native won't be able to operate in Mantle Chain. `ReentrancyGuardTransient.sol` implements a `nonReentrant` modifier via utilizing Transient storage, hence `TSTORE` and `TLOAD` opcodes introduced in EIP-1153. This `nonReentrant` modifier is used in all main entry functions for the protocol:

- `deposit` and `redeem` for LP's,
- `addCollateral`, `removeCollateral`, `repay`, `settle` for Market Makers,
- `liquidate` for liquidators,
- `tradeRFQT` for swappers
- Some helper and admin functions in `NativeRouter`

However Mantle Chain does not support opcodes `TLOAD` and `TSTORE` as of date as can be seen from Unsupported Opcodes.

Hence all calls to these functions will revert because of invalid opcode usage.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

It is a vulnerability that will occur naturally.

Impact

Issue renders all contracts useless in one of the chains (Mantle) that protocol will be deployed.

PoC

No response

Mitigation

For Mantle deployment, utilize old `ReentrancyGuard` -that doesn't use Transient Storage Opcodes- in all contracts.

Discussion

Ethronaut

Thanks for pointing that out, the chains we currently support are: **ETH, BNB, Base, Arbitrum, and Berachain**, we have not deployed to Mantle.

Issue M-5: Unable to sell native token using external router

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/59

Found by

0xc0ffEE, 0xrex

Summary

Seller token is not converted to wrapped native token while swapping with external router. This can cause the trade failed

Root Cause

In the function NativeRouter::tradeRFQT(), if the seller token is native token, then it will be wrapped to be WETH. From here, it can be accepted that the seller token is WETH, not the native token because the contract is holding WETH. However, if the trade is using external router, the order's seller token is still not updated to be WETH.

```
        // ...
        if (isNativePool) {

↪  NativeRFQPool(payable(quote.pool)).tradeRFQT(effectiveSellerTokenAmount, quote);
        } else if (whitelistRouter[quote.pool]) {
            Orders.Order memory order = Orders.Order({
                id: 0, // not used
                signer: address(0), // not used
                buyer: quote.pool,
                seller: address(0), // not used
                buyerToken: quote.buyerToken,
@>                sellerToken: quote.sellerToken, // <<<<<<<<<< this is still
↪  native token with address 0x0
                buyerTokenAmount: quote.buyerTokenAmount,
                sellerTokenAmount: quote.sellerTokenAmount,
                deadlineTimestamp: quote.deadlineTimestamp,
                caller: msg.sender,
                quoteId: quote.quoteId
            });

            uint256 actualAmountOut = ExternalSwap.externalSwap(
                order, effectiveSellerTokenAmount, quote.recipient, address(this),
↪  quote.externalSwapCalldata
```

```
            );
            // ...
```

When the execution goes into the function ExternalSwap::externalSwap(), it will revert because it will make contract call to address `order.sellerToken`, which is actuall `address(0)`. As a result, the trade will be failed

```
    function externalSwap(
        Orders.Order memory order,
        uint256 flexibleAmount,
        address recipient,
        address payer,
        bytes memory fallbackCalldata
    ) internal returns (uint256 amountOut) {
        require(flexibleAmount > 0, ErrorsLib.ZeroAmount());
        require(order.deadlineTimestamp >= block.timestamp,
↪   ErrorsLib.OrderExpired());

        SwapState memory state;
        (state.buyerTokenAmount, state.sellerTokenAmount) =
↪   _calculateTokenAmount(flexibleAmount, order);

        // prepare token for external call
        if (payer != address(this)) {
@>            IERC20(order.sellerToken).safeTransferFrom(payer, address(this),
↪   state.sellerTokenAmount);
        }
@>        IERC20(order.sellerToken).safeIncreaseAllowance(order.buyer,
↪   state.sellerTokenAmount);

        uint256 routerTokenOutBalanceBefore =
↪   IERC20(order.buyerToken).balanceOf(address(this));
        uint256 recipientTokenOutBalanceBefore =
↪   IERC20(order.buyerToken).balanceOf(recipient);

        {
            // call to external contract
            (bool success,) = order.buyer.call(fallbackCalldata);

            require(success, ErrorsLib.ExternalCallFailed(order.buyer,
↪   bytes4(fallbackCalldata)));
        }

        {
            // assume the tokenOut is sent to "recipient" by external call directly
            uint256 recipientDiff = IERC20(order.buyerToken).balanceOf(recipient) -
↪   recipientTokenOutBalanceBefore;
            uint256 routerDiff = IERC20(order.buyerToken).balanceOf(address(this))
↪   - routerTokenOutBalanceBefore;
```

```
            // if routerDiff is more, router has the tokens, so router transfers it
↪  out to recipient
            if (recipientDiff < routerDiff) {
                IERC20(order.buyerToken).safeTransfer(recipient, routerDiff);
                amountOut = IERC20(order.buyerToken).balanceOf(recipient) -
↪  recipientTokenOutBalanceBefore;
            } else {
                // otherwise, recipient has the tokens, so we can use recipientDiff
                amountOut = recipientDiff;
            }

            // amountOut is always the difference in after - before of recipient
↪  balance, to account for fee on transfer tokens
            require(amountOut >= state.buyerTokenAmount,
↪  ErrorsLib.NotEnoughTokenReceived());
        }

        emit ExternalSwapExecuted(
            order.buyer,
            order.caller,
            order.sellerToken,
            order.buyerToken,
            int256(state.sellerTokenAmount),
            -int256(amountOut),
            order.quoteId
        );
    }
```

Internal Pre-conditions

NA

External Pre-conditions

NA

Attack Path

1. An user requests to trade ETH -> USDC
2. The trade is quoted to use external router
3. User submits transaction and it reverts

Impact

- Unable to sell native token using external router

PoC

No response

Mitigation

```
            Orders.Order memory order = Orders.Order({
                id: 0, // not used
                signer: address(0), // not used
                buyer: quote.pool,
                seller: address(0), // not used
                buyerToken: quote.buyerToken,
-                sellerToken: quote.sellerToken,
+                sellerToken: quote.sellerToken == address(0) ? WETH9 :
 ↪  quote.sellerToken,
                buyerTokenAmount: quote.buyerTokenAmount,
                sellerTokenAmount: quote.sellerTokenAmount,
                deadlineTimestamp: quote.deadlineTimestamp,
                caller: msg.sender,
                quoteId: quote.quoteId
            });
```

Discussion

Ethronaut

Hi, **this is expected behavior.** Our platform primarily supports **RFQ** trading, while external swaps (i.e., AMMs) serve o**nly as a fallback**. We only forward to external swaps when our RFQ cannot provide a quote for certain trading pairs, or when the quotes from AMMs are better than ours.

Most AMM-based DEXs only support ERC-20 tokens – we do not automatically wrap the swapper's native tokens into ERC-20 tokens like WETH. So if the final AMM trade fails (e.g., due to not meeting the minimum output amount), the swapper bears the greater loss.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Native-org/v2-core/commit/6745b1deb50eda266ebcc4d724cff0c79448df83

Issue M-6: Tokens will be stuck in NativeRouter in a multihop swap with non-native pool

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/61

Found by

montecristo

Summary

`actualAmountOut` can not be pre-determined if a swap is done via a non-native pool (i.e. ExternalSwap).

If a multihop swap involves a non-native pool, neither `actualSellerAmount` or `quote.sellerTokenAmount` can be pre-calculated beforehand. The best thing we can do is to estimate the amount, but there will always be some slippage between `actuallSellerAmount` and `actualAmountOut`.

However, NativeRouter does not adjust `effectiveSellerTokenAmount` based on *previous* `actualAmountOut`.

As a result, either multihop swap reverts due to insufficient ERC20 balance, or some tokens will be stuck in NativeRouter contract.

Root Cause

Root cause is summarized in Summary section, but let's dive deeper into the problem with an example.

Let's assume the following:

- User wants to perform a swap WETH -> DAI -> WBTC with 20 ETH
- ETH -> DAI involves an external swap
- DAI -> WBTC uses a native pool
- All fees are 0, in order to make things simpler
- Quoter gives the user the following `RFQTQuotes`:
  - Quote 1
    * `pool`: address of Uniswap
    * `recipient`: address of NativeRouter
    * `sellerTokenAmount`: 20 WETH

    * `buyerTokenAmount`: 50000 DAI
    * `multiHop`: false
  - Quote 2
    * `pool`: address of DAI/WBTC native pool
    * `recipient`: `msg.sender`
    * `sellerTokenAmount`: 49500 DAI (to acknowledge DEX slippage)
    * `buyerTokenAmount`: 0.495 BTC
    * `multiHop`: true

Now the user will multicall NativeRouter to execute the following methods in a single transaction:

- NativeRouter::tradeRFQT(Quote1, 0, 0)
- NativeRouter::tradeRFQT(Quote2, 0, 0)

In the first trade, NativeRouter will swap 20 WETH to DAI. Let's assume Uniswap returned 49900 DAI for 20 WETH. In the second trade, NativeRouter will only use 49500 DAI to perform swap to WBTC. So 400 DAI will be stuck in NativeRouter contract.

The problem here is that no one can correctly estimate `actualAmountOut`. There will always be difference between `actualAmountOut` and `estimatedAmountOut`. And the difference will be stuck in NativeRouter contract.

Internal Pre-conditions

User performs a multihop swap that involves a non-native pool.

External Pre-conditions

n/a

Attack Path

n/a

Impact

Ineffective use of assets, users will face fund loss during a swap

PoC

No response

Mitigation

Mitigation is outlined in the following diff:

```
diff --git a/v2-core/src/NativeRouter.sol b/v2-core/src/NativeRouter.sol
index d5efc6c..404d8cb 100644
--- a/v2-core/src/NativeRouter.sol
+++ b/v2-core/src/NativeRouter.sol
@@ -81,6 +81,7 @@ contract NativeRouter is INativeRouter, EIP712, Ownable2Step,
↪  Pausable, Multical
     ) external payable override nonReentrant whenNotPaused {
         require(quote.widgetFee.feeRate <= MAX_WIDGET_FEE_BIPS,
↪  ErrorsLib.InvalidWidgetFeeRate());
         require(block.timestamp <= quote.deadlineTimestamp,
↪  ErrorsLib.QuoteExpired());
+        require(!quote.multiHop || actualSellerAmount == 0, "Cannot set amount in
↪  a multihop");

         _verifyRFQSignature(quote);

@@ -102,6 +103,8 @@ contract NativeRouter is INativeRouter, EIP712, Ownable2Step,
↪  Pausable, Multical
             require(deviation < MAX_AMOUNT_DEVIATION_BPS, "actual amount deviation
↪  exceeds 10%");

             effectiveSellerTokenAmount = actualSellerAmount;
+        } else if (quote.multiHop) {
+            effectiveSellerTokenAmount =
↪  IERC20(quote.sellerToken).balanceOf(address(this));
         }
```

Discussion

Ethronaut

Hi, `multi-hop` trades only occur within `NativePool` transactions, `externalswap` will never be used as one of multi-hop transaction

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Native-org/v2-core/commit/6745b1deb50eda266ebcc4d724cff0c79448df83

Issue M-7: OpenZeppelin v4.9.6 safeIncreaseAllowance incompatibility leads to USDT external swaps failure

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/80

Found by

0xaxaxa, 0xc0ffEE, 0xrex, Kose, dobrevaleri, iamandreiski, ifeco445, newspacexyz

Summary

The ExternalSwap::externalSwap() function uses `safeIncreaseAllowance()` from OpenZeppelin v4.9.6 (ref), which fails with certain tokens like USDT that revert on non-zero to non-zero approvals, causing all external swaps with such tokens to fail.

Root Cause

In ExternalSwap.sol::exterbalSwap() the code uses OpenZeppelin's `safeIncreaseAllowance()` function which is incompatible with tokens like USDT that have special approval behavior.

```
IERC20(order.sellerToken).safeIncreaseAllowance(order.buyer,
↪  state.sellerTokenAmount);
```

USDT is a "weird ERC20" token that reverts when trying to update an allowance from a non-zero value to another non-zero value. This is to prevent a specific front-running attack vector with ERC20 approvals.

The OpenZeppelin's SafeERC20 library in v4.9.6 uses `safeIncreaseAllowance()` which calls `approve()` with the old allowance + new value. For USDT, if any previous allowance exists (even 1 wei), this will cause the transaction to revert with "USDT approval failure."

This is problematic because the project documentation explicitly states that USDT will be supported:

> "Yes tokens will be integrated
> - Only whitelisted (admin approved) tokens will be added
> - Might include these "weird tokens": [...] 5. Approval Race Protections - like USDT"

Internal Pre-conditions

1. The protocol must whitelist USDT as a supported token

2. A user must attempt an external swap with USDT as the input token

External Pre-conditions

1. The `order.buyer` (external router) must have a non-zero allowance from the contract for USDT token

Attack Path

1. A user wants to trade USDT using an external swap through the protocol
2. The user calls `NativeRouter::tradeRFQT()` with USDT as the seller token
3. The router executes `ExternalSwap::externalSwap()`
4. The function tries to call `safeIncreaseAllowance()` on USDT
5. If any previous allowance exists (even 1 wei), the USDT contract will revert on the approval attempt
6. The transaction fails, preventing users from conducting external swaps with USDT

Impact

All external swaps that use USDT as the input token will fail, rendering a core protocol functionality unusable for one of the most widely used stablecoins. This directly contradicts the project's stated goal of supporting USDT and similar tokens with approval race protections.

PoC

1. A user initiates an external swap with USDT through `NativeRouter::tradeRFQT()`
2. The router calls `ExternalSwap::externalSwap()`, which attempts to move USDT tokens from the payer to the contract
3. Assuming the contract already has some USDT allowance (from a previous uncompleted swap), it then calls:

```
IERC20(order.sellerToken).safeIncreaseAllowance(order.buyer,
↪  state.sellerTokenAmount);
```

4. The `safeIncreaseAllowance()` function in OpenZeppelin's SafeERC20 (v4.9.6) executes:

```
function safeIncreaseAllowance(IERC20 token, address spender, uint256 value)
↪  internal {
    uint256 oldAllowance = token.allowance(address(this), spender);
```

```
    _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
↪   spender, oldAllowance + value));
}
```

5. USDT reverts on the approval call because it prevents changing an allowance from a non-zero value to another non-zero value

Mitigation

Upgrade OpenZeppelin to the latest version. The latest version of OpenZeppelin Contracts has better handling for problematic tokens like USDT.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Native-org/v2-core/commit/1df2de81c052cd6ebfdc23b57046e27ae4f49384

Issue M-8: buyerToken and sellerToken Fields Mutated Before Signature Verification, Causing Legitimate Quotes to Fail in In NativeRFQPool.tradeRFQT()

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/86

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

0xAadi, Kirkeelee, moray5554

Summary

In `NativeRFQPool.tradeRFQT()` Mutating the `buyerToken` and `sellerToken` fields in the `quote` struct before signature verification will cause valid off-chain signatures to fail, as the data used for verification no longer matches the original signed message. This breaks and prevents legitimate RFQ trades.

Root Cause

In the `tradeRFQT()` function, the following lines appear before the `_verifyPMMSignature()` call:

```
    function tradeRFQT(uint256 effectiveSellerTokenAmount, RFQTQuote memory quote)
↪   external override onlyRouter {
        // Prevent replay attacks
        require(!nonces[quote.nonce], ErrorsLib.NonceUsed());

        // Mark nonce as used
        nonces[quote.nonce] = true;

        // Store original buyerToken address to handle ETH unwrapping if buyerToken
↪   is zero address
        address originalBuyerToken = quote.buyerToken;

        // Handle ETH case: convert zero address to WETH9 for buyer or seller token
@>      quote.buyerToken = quote.buyerToken == address(0) ? WETH9 :
↪   quote.buyerToken;
@>       quote.sellerToken = quote.sellerToken == address(0) ? WETH9 :
↪   quote.sellerToken;

        // Verify market maker signature
```

```
@>      _verifyPMMSignature(quote);
```

https://github.com/sherlock-audit/2025-05-native-smart-contract-v2/blob/main/v2-core/src/NativeRFQPool.sol#L87C2-L102C36

This mutates the `quote` data, replacing `address(0)` with `WETH9`, before the `signature` is verified using EIP-712. As a result, the hash computed for signature recovery no longer reflects the original signed data. If a market maker signed a `quote` using `address(0)` to represent ETH, the on-chain verification will fail because the `quote.buyerToken` or `quote.sellerToken` has already been modified to `WETH9`.

EIP-712 signature verification requires the exact same data to be used when computing the hash. Any in-place mutation of struct fields before signature verification breaks this guarantee.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. Market maker signs an RFQ quote off-chain with `buyerToken` or `sellerToken` set to `address(0)` to indicate native ETH.
2. The `NativeRouter` submits the quote to `NativeRFQPool.tradeRFQT()`.
3. The contract replaces `address(0)` with `WETH9` in `quote.buyerToken` and `quote.sellerToken`.
4. The modified quote is used to compute the EIP-712 signature hash.
5. Signature verification fails because the hash differs from what was signed off-chain.
6. The transaction reverts, making the valid off-chain quote unusable on-chain.

Impact

This issue will cause valid off-chain RFQ signatures to fail verification when they contain `address(0)` for `buyerToken` or `sellerToken`. This prevents legitimate trades from being executed and could result in unnecessary quote rejections.

PoC

No response

Mitigation

```
    function tradeRFQT(uint256 effectiveSellerTokenAmount, RFQTQuote memory quote)
 ↪  external override onlyRouter {
        // Prevent replay attacks
        require(!nonces[quote.nonce], ErrorsLib.NonceUsed());

        // Mark nonce as used
        nonces[quote.nonce] = true;

        // Store original buyerToken address to handle ETH unwrapping if buyerToken
 ↪  is zero address
        address originalBuyerToken = quote.buyerToken;

+       // Verify market maker signature
+       _verifyPMMSignature(quote);

        // Handle ETH case: convert zero address to WETH9 for buyer or seller token
        quote.buyerToken = quote.buyerToken == address(0) ? WETH9 :
 ↪  quote.buyerToken;
        quote.sellerToken = quote.sellerToken == address(0) ? WETH9 :
 ↪  quote.sellerToken;

-       // Verify market maker signature
-       _verifyPMMSignature(quote);
```

Discussion

Ethronaut

Hi, this is not an issue

In the RFQTQuote struct, we have two different signature fields:

1: market maker's EIP-712 signature

2: Native backend signed widgetFeeSignature

It's clear from this structure: when a swap involves a `native token` as either the `buyToken` or `sellToken`, `the native backend treats such tokens as address(0)`. However, **our market makers always represented native tokens using their wrapped token addresses (e.g., WETH instead of address(0)).**

So, in _verifyRFQSignature, **quote.sellerToken and quote.buyerToken remain address(0)**, because that's how the native backend signed them.

But in _verifyPMMSignature, **these addresses are replaced with WETH**, since that's how the market maker signed them – by treating native tokens as their wrapped equivalents.

It's like a key and a lock – both must match. Each party (backend and market maker) signs according to their own interpretation, but both signatures are compliant with the EIP-712 standard.

Please don't misunderstand – this has nothing to do with whether the market maker is a whitelisted signer or not.

Issue M-9: Native swap transactions will revert in NativeRouter on new chains with strict WETH9 implementations

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/116

Found by

0xc0ffEE, Kirkeelee, Kose, eeyore, iamandreiski, jasonxiale, montecristo

Summary

The `_chargeWidgetFee()` function in `NativeRouter` can revert when used with native-wrapped tokens (e.g., WETH on Arbitrum, wBERA on Berachain, wMNT on Mantle) due to stricter `transferFrom()` allowance checks.

Vulnerability Detail

These WETH variants do not bypass the allowance check when `src == msg.sender`. Thus, if the user pays with native token (ETH, BERA, MNT) and the router wraps it, calling `safeTransferFrom(msg.sender, feeRecipient)` without an approved allowance will revert.

For native payments, `NativeRouter` uses the call `effectiveSellerTokenAmount = _chargeWidgetFee(widgetFee, sellerTokenAmount, WETH9, true);`, passing `true` for the `hasAlreadyPaid` parameter in `_chargeWidgetFee()`:

```
if (fee > 0) {
@>  TransferHelper.safeTransferFrom( // will DoS for WETH on Arbitrum, wBERA on
↪   Berachain, and wMNT on Mantle in case of native payment
        sellerToken, hasAlreadyPaid ? address(this) : msg.sender,
↪   widgetFee.feeRecipient, fee
    );
    emit WidgetFeeTransfer(widgetFee.feeRecipient, widgetFee.feeRate, fee,
↪   sellerToken);
    amountIn -= fee;
}
```

Impact

DoS for RFQ trades involving native tokens (ETH, BERA, MNT) on Arbitrum, Berachain, and Mantle.

Code Snippet

https://github.com/sherlock-audit/2025-05-native-smart-contract-v2/blob/main/v2-core/src/NativeRouter.sol#L259 https://github.com/sherlock-audit/2025-05-native-smart-contract-v2/blob/main/v2-core/src/NativeRouter.sol#L282-L284

Tool Used

Manual Review

Recommendation

Use `safeTransfer()` if `hasAlreadyPaid == true`:

```
if (fee > 0) {
    if (hasAlreadyPaid) {
        TransferHelper.safeTransfer(sellerToken, widgetFee.feeRecipient, fee);
    } else {
        TransferHelper.safeTransferFrom(sellerToken, msg.sender,
↪   widgetFee.feeRecipient, fee);
    }
    emit WidgetFeeTransfer(widgetFee.feeRecipient, widgetFee.feeRate, fee,
↪   sellerToken);
    amountIn -= fee;
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Native-org/v2-core/commit/f265cb3691fb9d6d4dda35c5c04cf72971ce3dec

Issue M-10: Valid trades will fail due to incorrect slippage validation

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/124

Found by

0xaxaxa, John44, dobrevaleri, montecristo

Summary

When a trade occurs through an external swap `ExternalSwap.externalSwap` will be called. The issue is that this function will validate that the received amount is not less than the `buyerTokenAmount` even though a trade should be valid when the received amount is not below the `amountOutMinimum` specified by the initiator.

Root Cause

In `ExternalSwap.externalSwap:73` the received tokens must be more or equal to `buyerTokenAmount`, even though they should only be more or equal to `amountOutMinimum`:

https://github.com/sherlock-audit/2025-05-native-smart-contract-v2/blob/main/v2-core/src/libraries/ExternalSwap.sol#L73

Internal Pre-conditions

No internal pre-conditions needed.

External Pre-conditions

No external pre-conditions needed.

Attack Path

1. A user performs a trade swapping 1e18 of AssetA for 2e18 of AssetB.
2. They set the `amountOutMinimum` to 1.9e18.
3. A trade is performed through an external swap, however, due to price changes the buyer sends out only 1.99e18 of AssetB.

4. The trade should still be valid as 1.99e18 is higher than the minimum specified by the caller, however, the call reverts due to the fact that `externalSwap` will check that 1.99e18 >= 2e18.

Impact

Valid trades will revert preventing users from interacting with the router.

PoC

No response

Mitigation

Consider not validating whether `amountOut >= state.buyerTokenAmount` as the necessary validation is performed in `NativeRouter`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/Native-org/v2-core/commit/6745b1deb50eda266ebcc4d724cff0c79448df83

Issue M-11: Some collateral can be locked in the Credit Vault contract

Source: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2-judging/issues/135

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

0xaxaxa, dobrevaleri, montecristo, tobi0x18

Summary

The `NativeLPToken` is not only a yield-bearing token but also a rebasing token. However, when updating the collateral amount, it is never accounted for. As a result, some `NativeLPToken` can be locked in the `CreditVault` contract.

Root Cause

When adding collateral into the `CreditVault`, the collateral amount of the trader is increased at L319 and the `NativeLPToken` is transferred from the trader at L321.

```
    function addCollateral(TokenAmountUint[] calldata tokens, address trader)
↪   external nonReentrant {
        require(traders[trader], ErrorsLib.OnlyTrader());

        for (uint256 i; i < tokens.length; ++i) {
            address token = tokens[i].token;
            require(supportedMarkets[token], ErrorsLib.OnlyLpToken());

            uint256 amount = tokens[i].amount;
319:        collateral[trader][token] += amount;

321:        IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
        }

        emit CollateralAdded(trader, tokens);
    }
```

However, the `NativeLPToken` is not only a yield-bearing token but also a rebasing token.

https://github.com/sherlock-audit/2025-05-native-smart-contract-v2/blob/main/v2-core/src/CreditVault.sol#L422-L426

```
function _transfer(address from, address to, uint256 amount) internal override {
    uint256 sharesToTransfer = getSharesByUnderlying(amount);
    _transferShares(from, to, sharesToTransfer);
    _emitTransferEvents(from, to, amount, sharesToTransfer);
}
```

As a result, when removing collateral from the `CreditVault`, some shares will remain, if some yields have been generated. https://github.com/sherlock-audit/2025-05-native-smart-contract-v2/blob/main/v2-core/src/CreditVault.sol#L203-L225

```
function removeCollateral(
    RemoveCollateralRequest calldata request,
    bytes calldata signature
) external onlyTraderOrSettler(request.trader) nonReentrant {
    _verifyRemoveCollateralSignature(request, signature);

    for (uint256 i; i < request.tokens.length; ++i) {
        collateral[request.trader][request.tokens[i].token] -=
↪   request.tokens[i].amount;
    }

    address recipient = traderToRecipient[request.trader];
    for (uint256 i; i < request.tokens.length; ++i) {
        address token = request.tokens[i].token;
        uint256 amount = request.tokens[i].amount;

        /// Enforce rebalance cap before funds leave vault
        _updateRebalanceCap(request.trader, token, amount);

        IERC20(token).safeTransfer(recipient, amount);
    }

    emit CollateralRemoved(request.trader, request.tokens);
}
```

Internal pre-conditions

External pre-conditions

Attack Path

Impact

Some collateral can be locked in the `CreditVault` contract, preventing traders from withdrawing the accrued yield.

PoC

Mitigation

The amount of shares should be tracked instead of the amount of underlying tokens.

Discussion

Ethronaut

Hi, I think there's a misunderstanding about how rebasing tokens work. The trader's collateral is NativeLPToken. For example:

Trader Alice adds 1000 Native-USDC LP as collateral, which gives her 1000 shares. Let's assume the exchange rate is initially 1. Even after multiple rounds of epoch updates, then the exchange rate rises to 1.1, her share count remains 1000, not 1100.

Therefore, when `removeCollateral` is called, it only transfers the shares, not the underlying amount The underlying value will never be locked in the credit vault

WangSecurity

When the trader withdraws from the `CreditVault`, the `amount` can be up to the original `collateral[trader][token]`.

Since the `NativeLPToken` is an yield-bearing, rebasing token, if the exchange rate increases after `addCollateral` is called, for a withdrawal of size equal `amount`, less `shares` will be needed to fulfill it. (https://github.com/sherlock-audit/2025-05-native-smart-contract-v2/blob/0872a93073063d7c7a05a2226a04d22a5923a3fe/v2-core/src/NativeLPToken.sol allowbreak #L422-L423)

In the `CreditVault` contract, there is no direct method to sweep/rescue the remaining `NativeLPToken.shares(address(this))`.

Although the trader receives exactly the same amount of underlying as their original deposited amount, the remaining shares are left in the `CreditVault` contract.

For example, in this scenario:

> Trader Alice adds 1000 Native-USDC LP as collateral, which gives her 1000 shares. Let's assume the exchange rate is initially 1. Even after multiple rounds of epoch updates, then the exchange rate rises to 1.1, her share count remains 1,000, not 1,100.

- Alice gets 1,000 USDC worth of LP token back;
- But 100 shares are left owned by the `CreditVault`, worth 110 USDC now.

`transfer` will adjust the `shares` to transfer based on the requested `amount`, so the additional generated value will be left in the `CreditPool`.

There is a workaround to rescue these remainders, but it is very inconvenient and will corrupt some mappings too. That's why we decided the issue should be valid.

Ethronaut

@WangSecurity that's exactly where I think you might have misunderstood.

Whether the market maker is calling `addCollateral` or `removeCollateral`, it's always based on **the number of shares**, not the underlying amount.

This is because the NativeLP token is a rebasing token – meaning the same amount of shares can represent different amounts of underlying over time, so if we were to `removeCollateral` based on the underlying amount, it could easily leave a small residual amount of shares stuck in the contract.

you can also refer our redeem function – its parameters are all based on `sharesToBurn`, not the `underlying` amount.

WangSecurity

> Whether the market maker is calling addCollateral or removeCollateral, it's always based on the number of shares, not the underlying amount.

The current code utilises the `underlyingAmount`-based `_transfer` function (which is used under `safeTransfer` in `removeCollateral`, resulting in transfers that leave the accrued yield behind, equal to the quantity of the remaining shares.

For instance, if transferShares was used instead in `removeCollateral`, the problem would most likely not exist.

But, currently, it uses exactly the amount-based `_transfer`: https://github.com/sherlock-audit/2025-05-native-smart-contract-v2/blob/0872a93073063d7c7a05a2226a04d22a5923a3fe/v2-core/src/NativeLPToken.sol allowbreak #L422-L425

```
/// @notice Override ERC20's _transfer to handle yield-bearing LP token transfers
/// @dev Since this is a yield-bearing token, the actual transfer is done by
 ↪  transferring shares
///      rather than token amounts directly. The shares represent the user's
 ↪  proportion of the
///      total underlying assets including yield.
/// @param from The address to transfer from
/// @param to The address to transfer to
/// @param amount The underlying token amount to transfer
function _transfer(address from, address to, uint256 amount) internal override {
    uint256 sharesToTransfer = getSharesByUnderlying(amount);
    _transferShares(from, to, sharesToTransfer);
    _emitTransferEvents(from, to, amount, sharesToTransfer);
}
```

The Lead Judge also made a POC here https://gist.github.com/c-plus-plus-equals-c-plus-one/006f790bafb9e9a16020502b171d9683

In this PoC, when the `exchangeRate` grows by 10% and less `shares` are needed to fulfill the same `amount`, `1000000000 - 900000000 = 100000000` shares are locked in the `CreditVault`, while the trader's `collateral` mapping is `0`.

Therefore, the generated yield is locked. Let us know if we're misunderstanding something.

Ethronaut

@WangSecurity sorry, you are right, after double-checking, the `addCollateral` and `removeCollateral` do use `ERC20 transfer`, which means the operations are based on the underlying amount, not shares.

this could lead to the issue you described, we'll fix it.

Ethronaut

@WangSecurity After internal discussion„ we will adopt a model similar to Lido's wstETH

specifically, if a market maker wants to use the NativeLP token as collateral, they must wrap it into a non-rebasing token (wstNLP), otherwise, **we will only accept non-rebasing ERC-20 tokens as collatera**l the add/remove Collateral request need be signed from our backend)

Given this design decision, we will not fix this issue, please include this response in the final audit report, thanks

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.