Gnoswap

Security Assessment

Security Report Published by BigInteger

Published on : 25 Sept. 2024
Audit Start Date : 2024/08/12
Auditor : Andy, Louis

Summary

Severity	Findings	Resolved	Acknowledged	Comment
Critical	1	1	-	-
High	2	2	-	-
Medium	5	5	-	-
Low	8	7	-	1
Tips	3	2	-	1

CONTENTS

Executive Summary

Audit Overview and Focus Areas

This report was prepared to audit the security of the gnoswap contracts developed by the ONBLOC team. BigInteger conducted the audit focusing on whether the system created by the ONBLOC team is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the Gnoswap.

In detail, we have focused on the following

- Pool package: Validation of core functionalities such as Swap, Mint, and Burn
- Position package: Security assessment of functions such as Mint, IncreaseLiquidity, DecreaseLiquidity, and Reposition
- Router package: Review of the efficiency and security of the SwapRoute logic
- Staker package: Validation of functions such as StakeToken, MintAndStake, and CalculatePoolPosition
- GNS package: Verification of the implementation of functions such as Mint and SetAvgBlockTimeInMs
- Core libraries: Stability and performance testing of essential libraries such as u256 and i256

Findings

According to BigInteger's audit results, 1 Critical, 2 High, 5 Medium, and 8 Low severity issues were identified. Additionally, 3 suggestions for code improvement were categorized under the 'Tips' section.

#ID	Title	Type	Severity	Status
1	The function sqrtPriceMathGetNextSqrtPriceFromInput() does not trigger a panic when the price is 0.	Off Standard	Low	Resolved
2	The function sqrtPriceMathGetNextSqrtPriceFromInput() does not trigger a panic when the liquidity is 0.	Off Standard	Low	Resolved
3	The function sqrtPriceMathGetNextSqrtPriceFromInput() does not trigger a panic when an overflow occurs due to the input amount.	Off Standard	Low	Comment

4	The function sqrtPriceMathGetNextSqrtPriceFromOutput() does not trigger a panic when the liquidity is 0.	Off Standard	Low	Resolved
5	The StakeToken function allows staking without transferring token ownership.	Access & Privilege Control	Critical	Resolved
7	It is possible to create a wugnot-wugnot pool.	Input Validation	Tips	Resolved
8	The functions TickMathGetSqrtRatioAtTick and TickMathGetTickAtSqrtRatio do not check the range of the input values.	Input Validation	Medium	Resolved
9	The function TickMathGetTickAtSqrtRatio does not return a tick value of 0 when the input price is √(1/1)	Logic Error/Bug	High	Resolved
10	During the process of changing the Pool Path in the correct order, the sign of the tick value changes.⌙	Input Validation	Tips	Resolved
11	When a pool is created with a price lower than MIN_TICK_PRICE, it does not trigger a panic.	Input Validation	Low	Resolved
12	Minting 0 does not trigger a panic.	Off Standard	Low	Resolved
13	It is possible to mint beyond the MIN_TICK and MAX_TICK range.	Input Validation	Medium	Resolved
14	[informational] The router performs a separate approval from the user to collect fees.	N/A	Tips	Comment
15	Incorrect Reward Calculation in CreateExternalIncentive Function	Logic Error/Bug	High	Resolved
16	When swapping from native to GRC20, there is no comparison between the amount sent by the origin and the amountSpecified.	Input Validation	Low	Resolved
17	In the Int256 library, an edge case occurs due to the creation of -0.	Arithmetic	Medium	Resolved
18	When limitCaller is false, incorrect behavior of AllowCallFromOnly prevents function usage.	Denial of Service	Medium	Resolved
19	A single user can take all the Fees generated in a Pool.	Input Validation	Medium	Resolved
20	GNFT positions with duplicate TokenURIs can be created.	Logic Error/Bug	Low	Resolved

Scope

The audited codebase can be found on GitHub (https://github.com/gnoswap-labs/gnoswap) in the ‘beta_v2_audit_fix’ branch. The final version of the code reviewed in this audit corresponds to commit hash *7bcbf92683c0df0e75430605fe4440b721a1e9c8*.
The commit hash after the fix review is *023bb4dd5dd2267f7fd126fe39fca49256d62505.*

```
├── pool
│   ├── _GET_no_receiver.gno
│   ├── _GET_receiver.gno
│   ├── _RPC_api.gno
│   ├── _RPC_dry.gno
│   ├── emergency_halt.gno
│   ├── gno.mod
│   ├── liquidity_math.gno
│   ├── pool.gno
│   ├── pool_manager.gno
│   ├── position.gno
│   ├── position_modify.gno
│   ├── position_update.gno
│   ├── protocol_fee_pool_creation.gno
│   ├── protocol_fee_withdrawal.gno
│   ├── tick.gno
│   ├── tick_bitmap.gno
│   ├── token_register.gno
│   ├── type.gno
│   └── utils.gno
├── position
│   ├── _GET_no_receiver.gno.gno
│   ├── _GET_no_receiver_string.gno
│   ├── _RPC_api.gno
│   ├── _RPC_dry.gno
│   ├── gno.mod
│   ├── gno_helper.gno
│   ├── helper.gno
│   ├── liquidity_management.gno
│   ├── nft_helper.gno
│   ├── position.gno
│   ├── position_key.gno
│   ├── type.gno
│   ├── utils.gno
│   └── wrap_unwrap.gno
```

```
├── protocol_fee
│   ├── protocol_fee.gno
│   └── token_register.gno
├── router
│   ├── comptue_routes.gno
│   ├── gno.mod
│   ├── gno_helper.gno
│   ├── protocol_fee_swap.gno
│   ├── router.gno
│   ├── router_dry.gno
│   ├── swap_inner.gno
│   ├── swap_multi.gno
│   ├── swap_single.gno
│   ├── token_register.gno
│   ├── type.gno
│   ├── utils.gno
│   └── wrap_unwrap.gno
└── staker
    ├── _GET_no_receiver.gno
    ├── _RPC_api_incentive.gno
    ├── _RPC_api_stake.gno
    ├── calculate_pool_position_reward.gno
    ├── external_token_list.gno
    ├── gno.mod
    ├── gno_helper.gno
    ├── incentive_id.gno
    ├── manage_pool_tiers.gno
    ├── mint_stake.gno
    ├── protocol_fee_unstaking.gno
    ├── reward_math.gno
    ├── staker.gno
    ├── token_register.gno
    ├── type.gno
    ├── utils.gno
    ├── warm_up.gno
    ├── wrap_gns_block_time_change.gno
    └── wrap_unwrap.gno
```

FINDINGS

1. The function sqrtPriceMathGetNextSqrtPriceFromInput() does not trigger a panic when the price is 0.

ID: Gnoswap-01
Severity: Low
Type: Off Standard
Difficulty: Low
File: common/sqrt_price_math.gno

Issue

According to the spec, validation for the price should occur. However, the contract does not trigger a panic when the price is 0. Allowing a price of 0 can potentially lead to malfunction of the contract.

```
func sqrtPriceMathGetNextSqrtPriceFromInput(
        sqrtPX96 *u256.Uint, // uint160
        liquidity *u256.Uint, // uint128
        amountIn *u256.Uint, // uint256
        zeroForOne bool, // bool
) *u256.Uint { // uint160
        if zeroForOne {
                return sqrtPriceMathGetNextSqrtPriceFromAmount0RoundingUp(sqrtPX96, liquidity,
amountIn, true)
        }
        return sqrtPriceMathGetNextSqrtPriceFromAmount1RoundingDown(sqrtPX96, liquidity, amountIn,
true)
}
```

PoC

```
func TestGetNextSqrtPriceFromInput_1(t *testing.T) {
        // fails if price is zero
        sqrtPX96 = u256.Zero()
        liquidity = u256.Zero()
        amountIn = u256.MustFromDecimal("1000000000000000000") // 1e18
        zeroForOne = false
        amountIn.Div(amountIn, u256.NewUint(10))

        shouldPanic(
                t,
                func() {
                        common.SqrtPriceMathGetNextSqrtPriceFromInput(sqrtPX96, liquidity, amountIn,
zeroForOne)
                },
        )
}
```

Recommendation

It is recommended to add logic to validate the price value in the implementation to ensure its correctness.

Fix Comment

The code has been updated to trigger a panic when the sqrtPX96 value is 0.

Fix commit hash: [b7b805965322b67cc3564507344b5f439608e126]

```
func sqrtPriceMathGetNextSqrtPriceFromInput(
        sqrtPX96 *u256.Uint, // uint160
        liquidity *u256.Uint, // uint128
        amountIn *u256.Uint, // uint256
        zeroForOne bool, // bool
) *u256.Uint { // uint160
        if sqrtPX96.IsZero() {
                panic("sqrtPriceMathGetNextSqrtPriceFromInput_sqrtPX96 should not be zero")
        }
    ...
```

2. The function sqrtPriceMathGetNextSqrtPriceFromInput() does not trigger a panic when the liquidity is 0.

ID: Gnoswap-02
Severity: Low
Type: Off Standard
Difficulty: Low
File: common/sqrt_price_math.gno

Issue

According to the spec, validation for liquidity should occur. However, the contract does not trigger a panic when liquidity is 0. Allowing a liquidity of 0 can potentially lead to malfunction of the contract.

```
func sqrtPriceMathGetNextSqrtPriceFromInput(
        sqrtPX96 *u256.Uint, // uint160
        liquidity *u256.Uint, // uint128
        amountIn *u256.Uint, // uint256
        zeroForOne bool, // bool
) *u256.Uint { // uint160
        if zeroForOne {
                return sqrtPriceMathGetNextSqrtPriceFromAmount0RoundingUp(sqrtPX96, liquidity,
amountIn, true)
        }
        return sqrtPriceMathGetNextSqrtPriceFromAmount1RoundingDown(sqrtPX96, liquidity, amountIn,
true)
}
```

PoC

```
func TestGetNextSqrtPriceFromInput_2(t *testing.T) {
        // fails if liquidity is zero
        sqrtPX96 = u256.One()
        liquidity = u256.Zero()
        amountIn = u256.MustFromDecimal("1000000000000000000") // 1e18
        zeroForOne = true
        amountIn.Div(amountIn, u256.NewUint(10))

        shouldPanic(
                t,
                func() {
                        common.SqrtPriceMathGetNextSqrtPriceFromInput(sqrtPX96, liquidity, amountIn,
zeroForOne)
                },
        )
}
```

Recommendation

It is recommended to add logic to validate the `liquidity` value in the implementation to ensure its correctness.

Fix Comment

The code has been updated to trigger a panic when the liquidity value is 0.

Fix commit hash: [5715e8ca245084b848c6771e2690a4a2cd921c9a]

```
func sqrtPriceMathGetNextSqrtPriceFromInput(
        sqrtPX96 *u256.Uint, // uint160
        liquidity *u256.Uint, // uint128
        amountIn *u256.Uint, // uint256
        zeroForOne bool, // bool
) *u256.Uint { // uint160
        ...

        if liquidity.IsZero() {
                panic("sqrtPriceMathGetNextSqrtPriceFromInput_liquidity should not be zero")
        }
        ...
```

3. The function sqrtPriceMathGetNextSqrtPriceFromInput() does not trigger a panic when an overflow occurs due to the input amount.

ID: Gnoswap-03
Type: Off Standard
File: common/sqrt_price_math.gno

Severity: Low
Difficulty: Low

Issue

According to the spec, validation for the input amount should occur. However, the contract does not trigger a panic when an overflow occurs in the input amount. Allowing an overflow can potentially lead to malfunction of the contract.

```
func sqrtPriceMathGetNextSqrtPriceFromInput(
        sqrtPX96 *u256.Uint, // uint160
        liquidity *u256.Uint, // uint128
        amountIn *u256.Uint, // uint256
        zeroForOne bool, // bool
) *u256.Uint { // uint160
        if zeroForOne {
                return sqrtPriceMathGetNextSqrtPriceFromAmount0RoundingUp(sqrtPX96, liquidity,
amountIn, true)
        }
        return sqrtPriceMathGetNextSqrtPriceFromAmount1RoundingDown(sqrtPX96, liquidity, amountIn,
true)
}
```

PoC

```
func TestGetNextSqrtPriceFromInput_3(t *testing.T) {
        // fails if input amount overflows the price

        sqrtPX96 = u256.NewUint(2)
        sqrtPX96.Exp(sqrtPX96, u256.NewUint(160))
        sqrtPX96.Sub(sqrtPX96, u256.One())
        liquidity = u256.NewUint(1024)
        amountIn = u256.NewUint(1024)
        zeroForOne = false

        shouldPanic(
                t,
                func() {
                        common.SqrtPriceMathGetNextSqrtPriceFromInput(sqrtPX96, liquidity, amountIn,
zeroForOne)
                },
        )
}
```

Recommendation

It is recommended to add validation logic for overflow in the input amount to ensure the contract's proper functioning.

Comment from Auditor

It has been confirmed that Gonswap uses the u256 data type to store sqrt price, and the sqrtPriceMathGetNextSqrtPriceFromInput function is only used in the view function SwapMathComputeSwapStepStr. Therefore, it has been verified that modifications related to this issue are not necessary in the current state.

4. The function sqrtPriceMathGetNextSqrtPriceFromOutput() does not trigger a panic when the liquidity is 0.

ID: Gnoswap-04
Severity: Low
Type: Off Standard
Difficulty: Low
File: common/sqrt_price_math.gno

Issue

According to the spec, validation for liquidity should occur. However, the contract does not trigger a panic when liquidity is 0. Allowing a value of 0 can potentially cause malfunction in the contract.

```
func sqrtPriceMathGetNextSqrtPriceFromOutput(
        sqrtPX96 *u256.Uint, // uint160
        liquidity *u256.Uint, // uint128
        amountOut *u256.Uint, // uint256
        zeroForOne bool, // bool
) *u256.Uint { // uint160
        if zeroForOne {
                return sqrtPriceMathGetNextSqrtPriceFromAmount1RoundingDown(sqrtPX96, liquidity,
amountOut, false)
        }

        return sqrtPriceMathGetNextSqrtPriceFromAmount0RoundingUp(sqrtPX96, liquidity, amountOut,
false)
}
```

PoC

```
func TestGetNextSqrtPriceFromOutput_2(t *testing.T) {
        // fails if liquidity is zero
        sqrtPX96 = u256.One()
        liquidity = u256.Zero()
        amountOut = u256.MustFromDecimal("100000000000000000") // 1e18
        amountOut.Div(amountOut, u256.NewUint(10))
        zeroForOne = true

        shouldPanic(
                t,
                func() {
                        common.SqrtPriceMathGetNextSqrtPriceFromOutput(sqrtPX96, liquidity, amountOut,
zeroForOne)
                },
        )
}
```

Recommendation

It is recommended to add logic to validate the `liquidity` value in the implementation to ensure its correctness and prevent potential issues.

Fix Comment

The code has been updated to trigger a panic when the liquidity value is 0.

Fix commit hash: [cdf27edb848130b9c489d1c05099a38ed7047d82]

```
func SqrtPriceMathGetNextSqrtPriceFromOutput(
        sqrtPX96 *u256.Uint, // uint160
        liquidity *u256.Uint, // uint128
        amountOut *u256.Uint, // uint256
        zeroForOne bool, // bool
) *u256.Uint { // uint160
        if liquidity.IsZero() {
                panic("sqrtPriceMathGetNextSqrtPriceFromOutput_liquidity should not be zero")
        }

...
```

5. The StakeToken function allows staking without transferring token ownership.

ID: Gnoswap-05
Severity: Critical
Type: Access & Privilege Control
Difficulty: Low
File: staker/staker.gno

Issue

In the `StakeToken` function, if the owner of the token to be staked is the Staker Realm, an approval is made for the calling user.

```
func StakeToken(tokenId uint64) (string, string, string) { // poolPath, token0Amount, token1Amount
...

        if callerIsOwner { // if caller is owner, transfer NFT ownership to staker contract
                transferDeposit(tokenId, GetOrigPkgAddr())
        } else if stakerIsOwner { // if staker is owner, approve NFT to caller (≈one click staking)
                gnft.Approve(a2u(std.GetOrigCaller()), tid(tokenId))
        }

        token0Amount, token1Amount := getTokenPairBalanceFromPosition(tokenId)
...
}
```

Due to this implementation, users can receive rewards without locking their GNFT and are able to change the liquidity value of the staked GNFT.

PoC

```
func TestStakeAndGetBack(t *testing.T) {
        //================ Create Pool =================
        std.TestSetRealm(gsaRealm)
        gns.Approve(a2u(consts.POOL_ADDR), pl.GetPoolCreationFee())
        pl.CreatePool(barPath, quxPath, 500, "130621891405341611593710811006")
        std.TestSkipHeights(1)

        //================ Mint Position =================
        bar.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
        qux.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
        std.TestSkipHeights(2)
        lpTokenId, liquidity, amount0, amount1 := pn.Mint(
                barPath,      // token0
                quxPath,      // token1
                uint32(500),  // fee
                int32(9000),  // tickLower
                int32(11000), // tickUpper
                "1000",       // amount0Desired
                "1000",       // amount1Desired
                "1",          // amount0Min
                "1",          // amount1Min
                max_timeout,  // deadline
```

```
                gsa.String(),
        )
        std.TestSkipHeights(1)
        gnft.Approve(a2u(GetOrigPkgAddr()), tid(lpTokenId))
        std.TestSkipHeights(1)

        //================ Stake Token =================
        gnft.TransferFrom(a2u(gsa), a2u(consts.STAKER_ADDR), tid(lpTokenId))
        StakeToken(lpTokenId)
        gnft.TransferFrom(a2u(consts.STAKER_ADDR), a2u(gsa), tid(lpTokenId))
        owner := gnft.OwnerOf(tid(lpTokenId))
        if owner != a2u(gsa) {
                t.Errorf("expected owner to be %s, got %s", a2u(gsa), owner)
        }
}
```

Recommendation

If the specific logic does not rely on approval, it is recommended to remove the `Approve` call from the `StakeToken` implementation.

Fix Comment

We have confirmed the removal of the logic for approving GNFT.

Fix commit hash: [08d55be6c0831c20a295c6e71e0066b616833d74]

```
func StakeToken(tokenId uint64) (string, string, string) { // poolPath, token0Amount, token1Amount
        ...
        if callerIsOwner { // if caller is owner, transfer NFT ownership to staker contract
                gnft.TransferFrom(a2u(owner), a2u(consts.STAKER_ADDR), tid(tokenId))
        }

        ...
}
```

7. It is possible to create a wugnot-wugnot pool.

ID: Gnoswap-07
Severity: Tips
Type: Input Validation
Difficulty: Low
File: pool/pool_manager.gno

Issue

The `CreatePool` function is designed to trigger an error when `token0Path` and `token1Path` are the same, preventing the creation of a pool with identical tokens. However, by entering "gnot" as `token0Path` and "gno.land/r/demo/wugnot" as `token1Path`, this validation can be bypassed, allowing the creation of a wugnot:wugnot pool.

```
func CreatePool(
        token0Path string,
        token1Path string,
        fee uint32,
        _sqrtPriceX96 string, // uint256
) {
...

        if token0Path == token1Path {
                panic(ufmt.Sprintf("[POOl] pool_manager.gno__CreatePool() || expected token0Path(%s) !=
token1Path(%s)", token0Path, token1Path))
        }

        if token0Path == consts.GNOT {
                token0Path = consts.WRAPPED_WUGNOT
        } else if token1Path == consts.GNOT {
                token1Path = consts.WRAPPED_WUGNOT
        }

...
}
```

PoC

```
func TestCreateWUGNOTWUGNOTPool(t *testing.T) {
        t.Run("TestCreateWUGNOTWUGNOTPool", func(t *testing.T) {
                std.TestSetOrigCaller(consts.GNOSWAP_ADMIN)
                std.TestSetRealm(std.NewUserRealm(std.Address((a2u(std.GetOrigCaller())))))
                gns.Approve(a2u(consts.POOL_ADDR), poolCreationFee)
                std.TestSetRealm(gsaRealm)

                CreatePool("gnot", "gno.land/r/demo/wugnot", fee500, "130621891405341611593710811006")
// tick 10000
                // --       event: {GNOSWAP gno.land/r/gnoswap/v2/pool CreatePool [{m_callType DIRECT}
{m_origCaller g1lmvrrrr4er2us84h2732sru76c9zl2nvknha8c} {m_prevRealm } {p_poolPath
gno.land/r/onbloc/bar:gno.land/r/onbloc/foo:500}]}       shouldEQ(t, len(pools), 1)
                std.TestSkipHeights(1)

                println(DoesPoolPathExist("gno.land/r/demo/wugnot:gno.land/r/demo/wugnot:500"))
```

```
        })
}
```

Recommendation

It is recommended to modify the logic so that tokens are replaced with `WRAPPED_WUGNOT` when the token is `GNOT` before performing the duplicate token check.

Fix Comment

The logic has been updated so that when the token is GNOT, it is replaced with WRAPPED_WUGNOT before performing the token duplication check.

Fix commit hash: [c3cdf6c203cedae93b4b166fda42305e5d032229]

```
func CreatePool(
        token0Path string,
        token1Path string,
        fee uint32,
        _sqrtPriceX96 string, // uint256
) {
        en.MintAndDistributeGns()

        // wrap first
        if token0Path == consts.GNOT {
                token0Path = consts.WRAPPED_WUGNOT
        } else if token1Path == consts.GNOT {
                token1Path = consts.WRAPPED_WUGNOT
        }

        // then check if token0Path == token1Path
        if token0Path == token1Path {
                panic(ufmt.Sprintf("[POOl] pool_manager.gno__CreatePool() || expected token0Path(%s) !=
token1Path(%s)", token0Path, token1Path))
        }

...
}
```

8. The functions TickMathGetSqrtRatioAtTick and TickMathGetTickAtSqrtRatio do not check the range of the input values.

ID: Gnoswap-08
Severity: Medium
Type: Input Validation
Difficulty: Low
File: common/tick_math.gno

Issue

The `TickMathGetSqrtRatioAtTick` and `TickMathGetTickAtSqrtRatio` functions convert a square root price to a tick and a tick to a square root price, respectively. These functions may return incorrect results for input values outside the defined range, so it is essential to validate the input values within the acceptable range.

```
func TickMathGetSqrtRatioAtTick(tick int32) *u256.Uint { // uint160 sqrtPriceX96
        absTick := abs(tick)
        ratio := u256.MustFromDecimal("340282366920938463463374607431768211456") // consts.Q128

        for mask, value := range tickRatioMap {
                if absTick&mask != 0 {
                        // ratio = (ratio * value) >> 128
                        ratio = ratio.Mul(ratio, value)
                        ratio = ratio.Rsh(ratio, 128)
                }
        }

        if tick > 0 {
                _maxUint256 :=
u256.MustFromDecimal("115792089237316195423570985008687907853269984665640564039457584007913129639935")
// consts.MAX_UINT256
                _tmp := new(u256.Uint).Div(_maxUint256, ratio)
                ratio = _tmp.Clone()
        }

        shifted := ratio.Rsh(ratio, 32).Clone()

        remainder := ratio.Mod(ratio, shift1By32Left)

        if new(u256.Uint).Add(shifted.Clone(), remainder.Clone()).IsZero() {
                return shifted
        }

        return new(u256.Uint).Add(shifted, u256.One())
}
```

```
func TickMathGetTickAtSqrtRatio(sqrtPriceX96 *u256.Uint) int32 {

        ratio := new(u256.Uint).Lsh(sqrtPriceX96, 32)

        msb, adjustedRatio := findMSB(ratio)
```

```
        adjustedRatio = adjustRatio(ratio, msb)

        log2 := calculateLog2(msb, adjustedRatio)
        tick := getTickValue(log2, sqrtPriceX96)

        return tick
}
```

This could allow minting beyond the `MIN_TICK` and `MAX_TICK` range and may also fail to trigger a panic when creating a pool with a price that is too low.

PoC

```
func TestTickMathGetSqrtRatioAtTick_1(t *testing.T) {
        // throws for too low
        var tick int32 = MIN_TICK
        var sqrtPriceX96 *u256.Uint

        tick = tick - 1

        shouldPanic( // does not check the abs(tick)
                t,
                func() {
                        common.TickMathGetSqrtRatioAtTick(tick)
                },
        )
}
```

```
func TestTickMathGetSqrtRatioAtTick_2(t *testing.T) {
        // throws for too low
        var tick int32 = MAX_TICK
        var sqrtPriceX96 *u256.Uint

        tick = tick + 1

        shouldPanic( // does not check the abs(tick)
                t,
                func() {
                        common.TickMathGetSqrtRatioAtTick(tick)
                },
        )
}
```

```
func TestTickMathGetTickAtSqrtRatio_1(t *testing.T) {
        // throws for too low
        var sqrtPriceX96 *u256.Uint = u256.MustFromDecimal(consts.MIN_SQRT_RATIO)
        sqrtPriceX96.Sub(sqrtPriceX96, u256.One())
        var rst int32

        shouldPanic( // does not check the range of price
                t,
                func() {
                        common.TickMathGetTickAtSqrtRatio(sqrtPriceX96)
                },
        )
```

```
}
```

```
func TestTickMathGetTickAtSqrtRatio_2(t *testing.T) {
        // throws for too high
        var sqrtPriceX96 *u256.Uint = u256.MustFromDecimal(consts.MAX_SQRT_RATIO)
        var rst int32

        shouldPanic( // does not check the range of price
                t,
                func() {
                        common.TickMathGetTickAtSqrtRatio(sqrtPriceX96)
                },
        )
}
```

Recommendation

It is recommended to add validation logic to verify the range of the input `tick` and `price` values to ensure correctness and prevent potential issues.

Fix Comment

The logic has been updated to validate the range of the input tick and price.

Fix commit hash: [9f4de388b3201f676589059f61d5f6eaf557be9e]

```
func TickMathGetSqrtRatioAtTick(tick int32) *u256.Uint { // uint160 sqrtPriceX96
        absTick := abs(tick)
        if absTick > 887272 { // MAX_TICK
                panic("tick is out of range (larger than 887272)")
        }
...
}
```

```
func TickMathGetTickAtSqrtRatio(sqrtPriceX96 *u256.Uint) int32 {
        cond1 := sqrtPriceX96.Gte(u256.MustFromDecimal("4295128739"))
// MIN_SQRT_RATIO
        cond2 :=
sqrtPriceX96.Lt(u256.MustFromDecimal("1461446703485210103287273052203988822378723970342")) //
MAX_SQRT_RATIO
        if !(cond1 && cond2) {
                panic("sqrtPriceX96 is out of range")
        }
...
}
```

9. The function TickMathGetTickAtSqrtRatio does not return a tick value of 0 when the input price is √(1/1)

ID: Gnoswap-09
Severity: High
Type: Logic Error/Bug
Difficulty: Medium
File: common/tick_math.gno

Issue

The `TickMathGetTickAtSqrtRatio` function calls `getTickValue` at the end to approximate the tick value.

```
func TickMathGetTickAtSqrtRatio(sqrtPriceX96 *u256.Uint) int32 {
        ...
        ratio := new(u256.Uint).Lsh(sqrtPriceX96, 32)

        msb, adjustedRatio := findMSB(ratio)
        adjustedRatio = adjustRatio(ratio, msb)

        log2 := calculateLog2(msb, adjustedRatio)
        tick := getTickValue(log2, sqrtPriceX96)

        return tick
}
```

```
func getTickValue(log2 *i256.Int, sqrtPriceX96 *u256.Uint) int32 {
...

        var tick int32
        if tickLow == tickHi {
                tick = tickLow
        } else if TickMathGetSqrtRatioAtTick(tickHi).Lte(sqrtPriceX96) {
                tick = tickHi
        } else {
                tick = tickLow
        }

        return tick
}
```

In the `getTickValue` function, when the square root price has a remainder within the 2^32 range, it performs a round-up by adding 1. However, instead of correctly checking if the remainder is within the 2^32 range by evaluating `((1 << 32) == 0 ? 0 : 1)`, the implementation incorrectly checks if the entire value is 0. This leads to an unintended outcome where, instead of returning the correct value of `79228162514264337593543950336`, an additional 1 is added, resulting in `79228162514264337593543950337`. Consequently, the `getTickAtSqrtRatio` function returns a tick value of -1 instead of 0.

This incorrect logic should be revised to properly check the remainder and ensure the accurate tick value is returned.

```
func TickMathGetSqrtRatioAtTick(tick int32) *u256.Uint { // uint160 sqrtPriceX96
        absTick := abs(tick)
        ratio := u256.MustFromDecimal("340282366920938463463374607431768211456") // consts.Q128

        for mask, value := range tickRatioMap {
                if absTick&mask != 0 {
                        // ratio = (ratio * value) >> 128
                        ratio = ratio.Mul(ratio, value)
                        ratio = ratio.Rsh(ratio, 128)
                }
        }

        if tick > 0 {
                _maxUint256 :=
u256.MustFromDecimal("115792089237316195423570985008687907853269984665640564039457584007913129639935")
// consts.MAX_UINT256
                _tmp := new(u256.Uint).Div(_maxUint256, ratio)
                ratio = _tmp.Clone()
        }

        shifted := ratio.Rsh(ratio, 32).Clone()

        remainder := ratio.Mod(ratio, shift1By32Left)

        if new(u256.Uint).Add(shifted.Clone(), remainder.Clone()).IsZero() {
                return shifted
        }

        return new(u256.Uint).Add(shifted, u256.One())
}
```

PoC

```
func TestTickMathGetTickAtSqrtRatio_Result(t *testing.T) {
        var rst int32
        var sqrtPriceX96 *u256.Uint

        ratios := []string{
                "4295128739",
                "79228162514264337593543950336000000",
                "7922816251426433759354395033600000",
                "99035203142830421991929937 92",
                "28011385487393069959365969113",
                "56022770974786139918731938227",
                "792281625142643375935439503 36",
                "11204554194957227983746387 6454",
                "22409108389914455967492775 2909",
                "63382530011411470074835160 2688",
                "79228162514264337593543950",
                "7922816251426433759354 3",
                "1461446703485210103287273052203988822378723970341",
        }

        expectedResults := []int32{
```

```
                -887272,
                276324,
                138162,
                -41591,
                -20796,
                -6932,
                0, // got -1, expected 0
                6931,
                20795,
                41590,
                -138163,
                -276325,
                887271,
        }

        for i, ratio := range ratios {
                sqrtPriceX96 = u256.MustFromDecimal(ratio)
                rst = common.TickMathGetTickAtSqrtRatio(sqrtPriceX96)
                shouldEQ(t, rst, expectedResults[i])
        }
}
```

Recommendation

It is recommended to modify the implementation to correctly check if the remainder falls within the 2^32 range by using `((1 << 32) == 0 ? 0 : 1)` to ensure proper rounding and accurate results.

Fix Comment

The logic has been updated to check whether the remainder value is within the range of 2^32 by verifying ((1<<32) == 0 ? 0 : 1).

Fix commit hash: [553e0971032f2eb3e75cbe30d66cd8d101915c85]

```
func TickMathGetSqrtRatioAtTick(tick int32) *u256.Uint { // uint160 sqrtPriceX96
        ...

        shifted := ratio.Rsh(ratio, 32).Clone()       // ratio >> 32
        remainder := ratio.Mod(ratio, shift1By32Left) // ratio % (1 << 32)

        var adj *u256.Uint
        if remainder.IsZero() {
                adj = u256.Zero()
        } else {
                adj = u256.One()
        }

        return new(u256.Uint).Add(shifted, adj)
}
```

10. During the process of changing the Pool Path in the correct order, the sign of the tick value changes.

ID: Gnoswap-10
Severity: Tips
Type: Input Validation
Difficulty: Low
File: pool/pool_manager.gno

Issue

When creating a pool, if `token0Path` and `token1Path` are not in the correct order, the function swaps the order and simultaneously changes the sign of the tick. This behavior can result in the pool being created with a tick value that the creator did not intend.

```
        if token1Path < token0Path {
                token0Path, token1Path = token1Path, token0Path
                tick := -(common.TickMathGetTickAtSqrtRatio(sqrtPriceX96))
                sqrtPriceX96 = common.TickMathGetSqrtRatioAtTick(tick)
        }
```

PoC

```
func TestCreateFooBarPool_can_be_init_at_MIN_SQRT_RATIO(t *testing.T) {
        std.TestSetRealm(gsaRealm)

        gns.Approve(a2u(consts.POOL_ADDR), poolCreationFee)

        token0Path := "gno.land/r/onbloc/foo1"

        pl.CreatePool(token0Path, barPath, 3000, "4295128739") // MIN_SQRT_RATIO
        poolPath := "gno.land/r/onbloc/bar:gno.land/r/onbloc/foo1:3000"
        poolTick := pl.PoolGetSlot0Tick(poolPath)
        shouldEQ(t, poolTick, -887272)
}
---
--- FAIL: TestCreateFooBarPool_can_be_init_at_MIN_SQRT_RATIO (0.01s)
got 887272, expected -887272
```

Recommendation

It is recommended to trigger a panic when `token0Path` and `token1Path` are not ordered correctly. Alternatively, explicitly document the logic where the tick sign changes to prevent unintended pool creation by the user.

Fix Comment

The logic has been updated to trigger a panic if token0Path and token1Path are not sorted.

Fix commit hash: [d899b9da73ce2d23046f1caa56fca503660fd663]

```
func CreatePool(
        token0Path string,
        token1Path string,
        fee uint32,
        _sqrtPriceX96 string, // uint256
) {
...

        // then check if token0Path == token1Path
        if token1Path < token0Path {
                panic(ufmt.Sprintf("[POOl] pool_manager.gno__CreatePool() || expected token0Path(%s) <
token1Path(%s)", token0Path, token1Path))
                // or we can adjust
                // token0Path, token1Path = token1Path, token0Path
                // tick := -(common.TickMathGetTickAtSqrtRatio(sqrtPriceX96))
                // sqrtPriceX96 = common.TickMathGetSqrtRatioAtTick(tick)
        }


...
}
```

11. When a pool is created with a price lower than MIN_TICK_PRICE, it does not trigger a panic.

ID: Gnoswap-11
Severity: Low
Type: Input Validation
Difficulty: Low
File: pool/pool_manager.gno

Issue

A panic should be triggered when attempting to create a pool with a price outside the Tick range, but this is not currently the case. This issue stems from Gnoswap-08. Therefore, once Gnoswap-08 is resolved, this issue will be resolved as well.

PoC

```
func TestCreateFooBarPool_Should_be_failed_if_price_is_too_low(t *testing.T) {
        // fails if starting price is too low
        shouldEQ(t, gns.TotalSupply(), 100000000000000)
        shouldEQ(t, gnsBalance(consts.EMISSION_ADDR), 0)
        shouldEQ(t, gnsBalance(consts.STAKER_ADDR), 0)
        shouldEQ(t, gnsBalance(consts.DEV_OPS), 0)

        std.TestSetRealm(gsaRealm)

        gns.Approve(a2u(consts.POOL_ADDR), poolCreationFee)

        shouldPanic(
                t,
                func() {
                        CreatePool(fooPath, barPath, 3000, "1")
                },
        )
}
```

Recommendation

It is recommended to resolve the Gnoswap-08 issue to address the related problems.

Fix Comment

Gnoswap-08 has been resolved.
Fix commit hash: [9f4de388b3201f676589059f61d5f6eaf557be9e]

12. Minting 0 does not trigger a panic.

ID: Gnoswap-12
Severity: Low
Type: Off Standard
Difficulty: Low
File: pool/pool.gno

Issue

According to the standard, minting 0 should trigger a panic. However, due to missing validation of the `_liquidityAmount`, minting 0 is allowed, which is not intended.

```
func Mint(
        token0Path string,
        token1Path string,
        fee uint32,
        recipient string,
        tickLower int32,
        tickUpper int32,
        _liquidityAmount string, // uint128
) (string, string) { // uint256 x2
        common.DisallowCallFromUser()
        common.AllowCallFromOnly(consts.POSITION_PATH)

        liquidityAmount := u256.MustFromDecimal(_liquidityAmount)

        pool := GetPool(token0Path, token1Path, fee)
        _, amount0, amount1 := pool.modifyPosition( // int256 x2
                ModifyPositionParams{
                        std.Address(recipient),            // owner
                        tickLower,                         // tickLower
                        tickUpper,                         // tickUpper
                        i256.FromUint256(liquidityAmount), // liquidityDelta
                },
        )

        if amount0.Gt(i256.Zero()) {
                pool.transferFromAndVerify(std.GetOrigCaller(), consts.POOL_ADDR, pool.token0Path,
amount0, true)
        }

        if amount1.Gt(i256.Zero()) {
                pool.transferFromAndVerify(std.GetOrigCaller(), consts.POOL_ADDR, pool.token1Path,
amount1, false)
        }

        /*
                std.Emit(
                        "GNOSWAP",
                        "m_callType", callType(),
                        "m_origCaller", origCaller(),
                        "m_prevRealm", prevRealm(),
                        "p_poolPath", GetPoolPath(token0Path, token1Path, fee),
                        "p_tickLower", int32ToStr(tickLower),
                        "p_tickUpper", int32ToStr(tickUpper),
                        "p_liquidityAmount", _liquidityAmount,
                        "amount0", amount0.ToString(),
```

```
                        "amount1", amount1.ToString(),
                )
        */

        return amount0.ToString(), amount1.ToString()
}
```

PoC

```
func TestMint_fail_if_amount_is_0(t *testing.T) {
        // ============================== Pool Setup ================================================
        std.TestSetRealm(gsaRealm)

        gns.Approve(a2u(consts.POOL_ADDR), poolCreationFee)

        token0Path := "gno.land/r/onbloc/foo9"

        foo9.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
        bar.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
        pl.CreatePool(barPath, token0Path, 3000, "2505414483750479311864138015 6") // encodePriceSqrt(1,
10)
        poolPath := "gno.land/r/onbloc/bar:gno.land/r/onbloc/foo9:3000"

        pl.Mint(token0Path, barPath, 3000, "g1ecely4gjy0yl6s9kt409ll330q9hk2lj9ls3ec", minTick,
maxTick, "3161")

        // ====================fails if total amount at tick exceeds the max=======================

        pool := GetPool(token0Path, barPath, 3000)
        tickSpacing := pool.tickSpacing
        shouldPanic(
                t,
                func() {
                        pl.Mint(token0Path, barPath, 3000, "g1ecely4gjy0yl6s9kt409ll330q9hk2lj9ls3ec",
minTick+tickSpacing, maxTick-tickSpacing, "0")
                },
        )
}
```

Recommendation

It is recommended to add validation logic for the input `liquidity` value to ensure correctness and prevent unintended behavior.

Fix Comment

The logic has been updated to add validation for liquidity, triggering a panic when 0 is minted.

Fix commit hash: [e2d2599f480e7641556426d9363a3da9b43325ed]

```
func Mint(
        token0Path string,
        token1Path string,
```

```
        fee uint32,
        recipient string,
        tickLower int32,
        tickUpper int32,
        _liquidityAmount string, // uint128
) (string, string) { // uint256 x2
...
        if liquidityAmount.IsZero() {
                panic("[POOL] pool.gno__Mint() || liquidityAmount == 0")
        }

...
}
```

13. It is possible to mint beyond the MIN_TICK and MAX_TICK range.

ID: Gnoswap-13
Severity: Medium
Type: Input Validation
Difficulty: Low
File: pool/pool_manager.gno

Issue

Minting beyond the `MIN_TICK` and `MAX_TICK` range is possible, which is a vulnerability stemming from Gnoswap-08.

PoC

```
func TestCheckPositionAboveAndBelowLimitTick(t *testing.T) {
        // ============================== Pool Setup ========================
        std.TestSetRealm(gsaRealm)
        std.TestSetOrigCaller(consts.GNOSWAP_ADMIN)

        gns.Approve(a2u(consts.POOL_ADDR), poolCreationFee)

        token0Path := "gno.land/r/onbloc/foo"

        foo.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
        bar.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)

        pl.CreatePool(barPath, token0Path, 3000, "2505414483750479311864138015 6") // encodePriceSqrt(1,
10)

        std.TestSkipHeights(1)
        poolPath := "gno.land/r/onbloc/bar:gno.land/r/onbloc/foo:3000"

        pool := GetPool(token0Path, barPath, 3000)
        tickSpacing := pool.tickSpacing
        minTick := (consts.MIN_TICK/tickSpacing)*tickSpacing - (tickSpacing * 2)
        maxTick := (consts.MAX_TICK/tickSpacing)*tickSpacing + (tickSpacing * 2)

        // ============================== Position Setup ================
        testaddr1 := testutils.TestAddress("test1")
        pl.Mint(token0Path, barPath, 3000, testaddr1.String(), minTick, maxTick, "3162")
        if r := recover(); r == nil {
                t.Errorf("should have panic")
        }

        std.TestSkipHeights(1)
        // ============================== Position Check ====================
        positionKey := ufmt.Sprintf("%s__%d__%d", testaddr1.String(), minTick, maxTick)
        positionKey = base64.StdEncoding.EncodeToString([]byte(positionKey))
        println(pool.positions[positionKey].liquidity.ToString())
}
```

Recommendation

It is recommended to resolve the Gnoswap-08 issue. Alternatively, you can address this issue by adding a function to validate the Tick range.

Fix Comment

Gnoswap-08 has been resolved. However, the test failed due to a bug in the Gnoswap VM.

Fix commit hash: 9f4de388b3201f676589059f61d5f6eaf557be9e

14. [informational] The router performs a separate approval from the user to collect fees.

ID: Gnoswap-14
Severity: Tips
Type: N/A
Difficulty: N/A
File: router/router.gno

Issue

The current swap process (1 hop) is as follows:

1. The user approves the inputToken to the pool.
2. The pool retrieves the inputToken from the user.
3. The pool sends the first swap outToken to the router.
4. The router approves the swap outToken to the pool.
5. The pool sends the second swap outToken to the user.
6. The router collects the fee from the user.

However, this method has the drawback that the user must approve the fee.

```
func handleSwapFee(
        outputToken string,
        amount *u256.Uint,
        isDry bool,
) *u256.Uint {
        if swapFee <= 0 {
                return amount
        }

        feeAmount := new(u256.Uint).Mul(amount, u256.NewUint(swapFee))
        feeAmount.Div(feeAmount, u256.NewUint(10000))
        feeAmountUint64 := feeAmount.Uint64()

        if !isDry {
                if outputToken == consts.GNOT { // unwrap if coin
                        // wugnot: buyer > router
                        transferFromByRegisterCall(outputToken, std.GetOrigCaller(),
consts.ROUTER_ADDR, feeAmountUint64)

                        // ugnot: wugnot > router
                        wugnot.Withdraw(feeAmountUint64)

                        // ugnot: router > feeCollector
                        banker := std.GetBanker(std.BankerTypeRealmSend)
                        banker.SendCoins(consts.ROUTER_ADDR, consts.PROTOCOL_FEE_ADDR,
std.Coins{{"ugnot", int64(feeAmountUint64)}})
                        std.Emit(
                                "GNOSWAP_PROTOCOL_FEE",
                                "m_callType", callType(),
                                "m_origCaller", origCaller(),
                                "m_prevRealm", prevRealm(),
                                "reason", "router_fee",
```

```
                                "token", "ugnot",
                                "amount", strconv.FormatUint(feeAmountUint64, 10),
                        )
                } else { // just transfer if grc20
                        ok := transferFromByRegisterCall(outputToken, std.GetOrigCaller(),
consts.PROTOCOL_FEE_ADDR, feeAmountUint64)
                        if !ok {
                                panic(ufmt.Sprintf("[ROUTER] protocol_fee_swap.gno__handleSwapFee() ||
expected transferFromByRegisterCall(%s, %s, %s, %d) == true", outputToken, std.GetOrigCaller(),
consts.PROTOCOL_FEE_ADDR, feeAmountUint64))
                        }
                        std.Emit(
                                "GNOSWAP_PROTOCOL_FEE",
                                "m_callType", callType(),
                                "m_origCaller", origCaller(),
                                "m_prevRealm", prevRealm(),
                                "reason", "router_fee",
                                "token", outputToken,
                                "amount", strconv.FormatUint(feeAmountUint64, 10),
                        )
                }
        }

        toUserAfterProtocol := new(u256.Uint).Sub(amount, feeAmount)
        return toUserAfterProtocol
}
```

The recommended logic is as follows:

1. The user approves the inputToken to the router.
2. The router retrieves the token from the user.
3. The pool retrieves the token from the router.
4. The pool sends the first swap outToken to the router.
5. The router approves the swap outToken to the pool.
6. The pool sends the second swap outToken to the router.
7. The router deducts the fee and sends the outToken to the user.

By implementing this proposed method, the router deducts the fee from the outToken before sending it to the user. As a result, the user no longer needs to approve the fee to the router, only needing to approve the inputToken, potentially saving some gas.

Comment from Auditor

We have confirmed the decision to retain the implementation in order to clearly record the tokens being exchanged and the fees associated with the swap.

15. Incorrect Reward Calculation in CreateExternalIncentive Function

ID: Gnoswap-15
Severity: High
Type: Logic Error/Bug
Difficulty: Low
File: staker/staker.gno

Issue

When adding an external incentive, if the `incentiveId` is the same, there is logic to add the `amount` to the existing value. However, because the calculation of `rewardPerBlockX96` only uses the added `amount`, a mismatch occurs between the `rewardAmount` and the calculated `rewardPerBlockX96`. In the code divided into two parts, part2 should be executed first, followed by part1. However, since part1 is executed first, `rewardPerBlockX96` is calculated to be less than the accumulated rewards.

```
// part1
                        incentiveDuration := endTimestamp - startTimestamp
                        incentiveBlock := incentiveDuration / consts.BLOCK_GENERATION_INTERVAL
                        rewardAmountX96 := new(u256.Uint).Mul(rewardAmount,
u256.MustFromDecimal(consts.Q96))
                        rewardPerBlockX96 := new(u256.Uint).Div(rewardAmountX96,
u256.NewUint(uint64(incentiveBlock)))
                        incentive.rewardPerBlockX96 = rewardPerBlockX96

// part2
                        incentive.rewardAmount = new(u256.Uint).Add(incentive.rewardAmount,
rewardAmount)
                        incentive.rewardLeft = new(u256.Uint).Add(incentive.rewardLeft, rewardAmount)
                        incentives[v] = incentive
```

PoC

```
func TestCreateExternalIncentive(t *testing.T) {
        //================ Create Pool =================
        std.TestSetRealm(gsaRealm)
        std.TestSetOrigCaller(gsa)
        gns.Approve(a2u(consts.POOL_ADDR), pl.GetPoolCreationFee())
        pl.CreatePool(barPath, bazPath, 500, common.TickMathGetSqrtRatioAtTick(0).ToString())

        gns.Approve(a2u(consts.POOL_ADDR), pl.GetPoolCreationFee())
        pl.CreatePool(fooPath, quxPath, 500, common.TickMathGetSqrtRatioAtTick(0).ToString())

        gns.Approve(a2u(consts.POOL_ADDR), pl.GetPoolCreationFee())
        foePath := "gno.land/r/onbloc/foe"
        pl.CreatePool(foePath, oblPath, 500, common.TickMathGetSqrtRatioAtTick(0).ToString())
        std.TestSkipHeights(1)

        //================ Create External Incentive =================
        barbazPath := "gno.land/r/onbloc/bar:gno.land/r/onbloc/baz:500"
```

```
        fooquxPath := "gno.land/r/onbloc/foo:gno.land/r/onbloc/qux:500"
        foeoblPath := "gno.land/r/onbloc/foe:gno.land/r/onbloc/obl:500"
        startTimeStamp := time.Now().AddDate(0, 0, 1).Truncate(24 * time.Hour).Unix()

        externalAmount := "1000000000"
        externalAmountU64 := uint64(1000000000)
        bar.Approve(a2u(consts.STAKER_ADDR), externalAmountU64)
        CreateExternalIncentive(
                barbazPath,
                barPath,
                externalAmount,
                startTimeStamp,
                startTimeStamp+TIMESTAMP_90DAYS,
        )

        foe.Approve(a2u(consts.STAKER_ADDR), externalAmountU64)
        CreateExternalIncentive(
                foeoblPath,
                foePath,
                externalAmount,
                startTimeStamp,
                startTimeStamp+TIMESTAMP_90DAYS,
        )

        externalAmount_1 := "999900000"
        externalAmountU64_1 := uint64(999900000)
        foo.Approve(a2u(consts.STAKER_ADDR), externalAmountU64)
        CreateExternalIncentive(
                fooquxPath,
                fooPath,
                externalAmount_1,
                startTimeStamp,
                startTimeStamp+TIMESTAMP_90DAYS,
        )

        externalAmount_2 := "100000"
        externalAmountU64_2 := uint64(100000)
        foo.Approve(a2u(consts.STAKER_ADDR), externalAmountU64)
        CreateExternalIncentive(
                fooquxPath,
                fooPath,
                externalAmount_2,
                startTimeStamp,
                startTimeStamp+TIMESTAMP_90DAYS,
        )

        //================ Mint Position =================
        user1Addr := std.DerivePkgAddr("user1.gno")
        user1Realm := std.NewUserRealm(user1Addr)
        user2Addr := std.DerivePkgAddr("user2.gno")
        user2Realm := std.NewUserRealm(user2Addr)
        user3Addr := std.DerivePkgAddr("user3.gno")
        user3Realm := std.NewUserRealm(user3Addr)
        tickSpacing := int32(10)
        minTick := (consts.MIN_TICK / tickSpacing) * tickSpacing
        maxTick := (consts.MAX_TICK / tickSpacing) * tickSpacing

        bar.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
        baz.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
        lpTokenId1, _, _, _ := pn.Mint(
                barPath,     // token0
                bazPath,     // token1
                uint32(500), // fee
```

```
        minTick,     // tickLower
        maxTick,     // tickUpper
        "1000000",   // amount0Desired
        "1000000",   // amount1Desired
        "1",         // amount0Min
        "1",         // amount1Min
        max_timeout, // deadline
        user1Addr.String(),
)

foo.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
qux.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
lpTokenId2, _, _, _ := pn.Mint(
        fooPath,     // token0
        quxPath,     // token1
        uint32(500), // fee
        minTick,     // tickLower
        maxTick,     // tickUpper
        "1000000",   // amount0Desired
        "1000000",   // amount1Desired
        "1",         // amount0Min
        "1",         // amount1Min
        max_timeout, // deadline
        user2Addr.String(),
)

foe.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
obl.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
lpTokenId3, _, _, _ := pn.Mint(
        foePath,     // token0
        oblPath,     // token1
        uint32(500), // fee
        minTick,     // tickLower
        maxTick,     // tickUpper
        "1000000",   // amount0Desired
        "1000000",   // amount1Desired
        "1",         // amount0Min
        "1",         // amount1Min
        max_timeout, // deadline
        user3Addr.String(),
)
std.TestSkipHeights(1)

//================ Stake GNFT Token =================
std.TestSetRealm(user1Realm)
std.TestSetOrigCaller(user1Addr)
gnft.Approve(a2u(consts.STAKER_ADDR), tid(lpTokenId1))
StakeToken(lpTokenId1)

std.TestSetRealm(user2Realm)
std.TestSetOrigCaller(user2Addr)
gnft.Approve(a2u(consts.STAKER_ADDR), tid(lpTokenId2))
StakeToken(lpTokenId2)

std.TestSetRealm(user3Realm)
std.TestSetOrigCaller(user3Addr)
gnft.Approve(a2u(consts.STAKER_ADDR), tid(lpTokenId3))
StakeToken(lpTokenId3)
std.TestSkipHeights(1)

//================ Collect Fee =================
std.TestSkipHeights((86400 / 5) * 30) // 1 height == 5 seconds
```

```
        user1BarBalanceBefore := bar.BalanceOf(a2u(user1Addr))
        user2FooBalanceBefore := foo.BalanceOf(a2u(user2Addr))
        user3FoeBalanceBefore := foe.BalanceOf(a2u(user3Addr))

        std.TestSetOrigCaller(user1Addr)
        std.TestSetRealm(user1Realm)
        CollectReward(lpTokenId1, false)

        std.TestSetOrigCaller(user2Addr)
        std.TestSetRealm(user2Realm)
        CollectReward(lpTokenId2, false)

        std.TestSetOrigCaller(user3Addr)
        std.TestSetRealm(user3Realm)
        CollectReward(lpTokenId3, false)

        user1BarBalanceAfter := bar.BalanceOf(a2u(user1Addr))
        user2FooBalanceAfter := foo.BalanceOf(a2u(user2Addr))
        user3FoeBalanceAfter := foe.BalanceOf(a2u(user3Addr))
        shouldEQ(
                t,
                user1BarBalanceAfter-user1BarBalanceBefore,
                user2FooBalanceAfter-user2FooBalanceBefore,
        )
        shouldEQ(
                t,
                user2FooBalanceAfter-user2FooBalanceBefore,
                user3FoeBalanceAfter-user3FoeBalanceBefore,
        )
        shouldEQ(
                t,
                user1BarBalanceAfter-user1BarBalanceBefore,
                user3FoeBalanceAfter-user3FoeBalanceBefore,
        )

        println("user1BarBalanceBefore: ", user1BarBalanceBefore)
        println("user2FooBalanceAfter: ", user2FooBalanceBefore)
        println("user3FoeBalanceBefore: ", user3FoeBalanceBefore)

        println("user1BarBalanceAfter: ", user1BarBalanceAfter)
        println("user2FooBalanceAfter: ", user2FooBalanceAfter)
        println("user3FoeBalanceAfter: ", user3FoeBalanceAfter)
}
----------------------------------------
user1BarBalanceBefore:  0
user2FooBalanceAfter:  0
user3FoeBalanceBefore:  0
user1BarBalanceAfter:  329692406
user2FooBalanceAfter:  32967
user3FoeBalanceAfter:  329692406
```

Recommendation

It is recommended to modify the logic so that the Part2 logic is executed before the Part1 logic, as described above. This will ensure that `rewardPerBlockX96` is calculated correctly and matches the accumulated reward amount.

Fix Comment

The logic has been modified so that the rewardPerBlockX96 value is calculated using the sum of the newly added amount and the existing amount.

Fix commit hash: [ebc3dd4ccd1fce576a02ecf7d883bb1326f1d2b4]

```
func CreateExternalIncentive(
        targetPoolPath string,
        rewardToken string, // token path should be registered
        _rewardAmount string,
        startTimestamp int64,
        endTimestamp int64,
) {
        ...
                        rewardAmountX96 := new(u256.Uint).Mul(rewardAmount,
u256.MustFromDecimal(consts.Q96))
                        rewardPerBlockX96 := new(u256.Uint).Div(rewardAmountX96,
u256.NewUint(uint64(incentiveBlock)))
                        incentive.rewardPerBlockX96 = rewardPerBlockX96

                        incentive.rewardAmount = new(u256.Uint).Add(incentive.rewardAmount,
rewardAmount)
                        incentive.rewardLeft = new(u256.Uint).Add(incentive.rewardLeft, rewardAmount)
                        incentive.depositGnsAmount += depositGnsAmount
                        incentives[v] = incentive

                        ...
}
```

16. When swapping from native to GRC20, there is no comparison between the amount sent by the origin and the amountSpecified.

ID: Gnoswap-16
Type: Input Validation
File: router/router.gno

Severity: Low
Difficulty: Low

Issue

In the case of swapping native tokens, the `SwapRout` function retrieves the amount of native tokens sent by the user using `std.GetOrigSend()`. However, it does not verify whether `ugnotSentByUser` and `_amountSpecified` are the same, which could potentially lead to unintended behavior.

```
func SwapRoute(
        inputToken string,
        outputToken string,
        _amountSpecified string, // int256
        swapType string,
        strRouteArr string, // []string
        quoteArr string, // []int
        _tokenAmountLimit string, // uint256
) (string, string) { // tokneIn, tokenOut
...

        var userBeforeWugnotBalance uint64
        var userWrappedWugnot uint64
        if inputToken == consts.GNOT || outputToken == consts.GNOT {
                userBeforeWugnotBalance = wugnot.BalanceOf(a2u(std.GetOrigCaller()))

                sent := std.GetOrigSend()
                ugnotSentByUser := uint64(sent.AmountOf("ugnot"))
                if ugnotSentByUser > 0 {
                        wrap(ugnotSentByUser)
                }
...

        return amountIn, amountOut
}
```

PoC

```
func TestSwapRouteWugnotquxExactInDifferentAmountCoinShouldPanic(t *testing.T) {
        std.TestSetRealm(gsaRealm)

        wugnot.Approve(a2u(consts.ROUTER_ADDR), 1000000)
        qux.Approve(a2u(consts.ROUTER_ADDR), 1000000)

        std.TestSetOrigSend(std.Coins{{"ugnot", 12345}}, nil) //sented ugnot amount
```

```
        shouldPanic(t,
                func() {
                        SwapRoute(
                                consts.GNOT, // inputToken
                                quxPath,     // outputToken
                                "3",         // amountSpecified   -> should be panic
                                "EXACT_IN",  // swapType
                                "gno.land/r/demo/wugnot:gno.land/r/onbloc/qux:3000", // strRouteArr
                                "100", // quoteArr
                                "1",   // tokenAmountLimit
                        )
                },
        )
}
```

Recommendation

It is recommended to add logic to verify that `ugnotSentByUser` and `_amountSpecified` are the same.

Fix Comment

The logic has been updated to include a comparison between the input amount and the amount of GNOT sent.

Fix commit hash: [3c0649356e6eee28367d13a0d135c46a7fdb8058]

```
func handleGNOT(inputToken, outputToken, _amountSpecified string) (uint64, uint64) {
        userOldWugnotBalance := uint64(0)
        if inputToken == consts.GNOT {
                sent := std.GetOrigSend()
                ugnotSentByUser := uint64(sent.AmountOf("ugnot"))

                i256AmountSpecified := i256.MustFromDecimal(_amountSpecified)
                u64AmountSpecified := i256AmountSpecified.Uint64()

                if ugnotSentByUser != u64AmountSpecified {
                        panic("[ROUTER] Invalid amount of ugnot sent by user")
                }

                wrap(ugnotSentByUser)
                userOldWugnotBalance = wugnot.BalanceOf(a2u(std.GetOrigCaller()))
        } else if outputToken == consts.GNOT {
...
```

17. In the Int256 library, an edge case occurs due to the creation of -0.

ID: Gnoswap-17
Type: Arithmetic
File: int256.gno

Severity: Medium
Difficulty: Medium

Issue

During arithmetic operations with int256, a negative zero value can be generated. While this edge case does not appear to affect the current Gnoswap implementation, it could lead to unintended results in certain comparisons or bitwise operations when this package is used in the future.

PoC

```
func TestInt256MinusZero(t *testing.T) {
        minusThree := i256.MustFromDecimal("-3")
        three := i256.MustFromDecimal("3")
        zero := i256.Zero()
        res := i256.Zero().Add(minusThree, three)
        println("-3 + 3 =",res.ToString())
        res2 := i256.Zero().Sub(minusThree, i256.MustFromDecimal("-3"))
        println("-3 - (-3) =", res2.ToString())

        res3 := i256.Zero().Sub(res, minusThree)
        res4 := i256.Zero().Sub(minusThree, res)
        res5 := i256.Zero().Sub(three, res)
        res6 := i256.Zero().Sub(res, three)

        res7 := i256.Zero().Add(res, minusThree)
        res8 := i256.Zero().Add(minusThree, res)
        res9 := i256.Zero().Add(three, res)
        res10 := i256.Zero().Add(res, three)

        res11 := i256.Zero().AddUint256(res, u256.MustFromDecimal("3"))
        res12 := u256.MustFromDecimal("0")
        i256.AddDelta(res12, u256.MustFromDecimal("3"), res)
        res13 := u256.MustFromDecimal("0")
        i256.AddDelta(res13, u256.MustFromDecimal("0"), res)
        res14 := i256.Zero().Mod(res, three)
        res15 := i256.Zero().Mod(res, minusThree)

        res16 := res.Eq(zero)
        res17 := res.Neq(zero)
        res18 := res.Cmp(zero)
        res19 := res.Lt(zero)
        res20 := res.Lt(three)
        res21 := res.Lt(minusThree)
        res22 := res.Gt(zero)
        res23 := res.Gt(three)
        res24 := res.Gt(minusThree)
        res25 := res.Cmp(three)
```

```
        res26 := res.Cmp(minusThree)

        res27 := i256.Zero().And(res, minusThree)
        res27_cmp := i256.Zero().And(zero, minusThree)
        res28 := i256.Zero().And(minusThree, res)
        res28_cmp := i256.Zero().And(minusThree, zero)
        res29 := i256.Zero().And(res, three)
        res29_cmp := i256.Zero().And(zero, three)
        res30 := i256.Zero().And(three,res)
        res30_cmp := i256.Zero().And(three,zero)

        res31 := i256.Zero().Or(res, minusThree)
        res31_cmp := i256.Zero().Or(zero, minusThree)
        res32 := i256.Zero().Or(minusThree, res)
        res32_cmp := i256.Zero().Or(minusThree, zero)
        res33 := i256.Zero().Or(res, three)
        res33_cmp := i256.Zero().Or(zero, three)
        res34 := i256.Zero().Or(three, res)
        res34_cmp := i256.Zero().Or(three, zero)

        res35 := i256.Zero().Rsh(res, 1234)
        res35_cmp := i256.Zero().Rsh(zero, 1234)
        res36 := i256.Zero().Lsh(res, 1234)
        res36_cmp := i256.Zero().Lsh(zero, 1234)

        shouldEQ(t, res3.ToString(), "3")
        shouldEQ(t, res4.ToString(), "-3")
        shouldEQ(t, res5.ToString(), "3")
        shouldEQ(t, res6.ToString(), "-3")
        shouldEQ(t, res7.ToString(), "-3")
        shouldEQ(t, res8.ToString(), "-3")
        shouldEQ(t, res9.ToString(), "3")
        shouldEQ(t, res10.ToString(), "3")
        shouldEQ(t, res11.ToString(), "3")
        shouldEQ(t, res12.ToString(), "3")
        shouldEQ(t, res13.ToString(), "0")
        shouldEQ(t, res14.ToString(), "0")
        shouldEQ(t, res15.ToString(), "0")

        shouldEQ(t, res16, true) // got false, expected true
        shouldEQ(t, res17, false) // got true, expected false
        shouldEQ(t, res18, 0) // got -1, expected 0
        shouldEQ(t, res19, false) // got true, expected false
        shouldEQ(t, res20, true)
        shouldEQ(t, res21, false)
        shouldEQ(t, res22, false)
        shouldEQ(t, res23, false)
        shouldEQ(t, res24, true)
        shouldEQ(t, res25, -1)
        shouldEQ(t, res26, 1)
        shouldEQ(t, res27.ToString(), res27_cmp.ToString()) // got -0, expected 0
        shouldEQ(t, res28.ToString(), res28_cmp.ToString()) // got -0, expected 0
        shouldEQ(t, res29.ToString(), res29_cmp.ToString())
        shouldEQ(t, res30.ToString(), res30_cmp.ToString())
        shouldEQ(t, res31.ToString(), res31_cmp.ToString())
        shouldEQ(t, res32.ToString(), res32_cmp.ToString())
        shouldEQ(t, res33.ToString(), res33_cmp.ToString()) // got
-1157920892373161954235709850086879078532699846656405640394575840079131296399333, expected 3
        shouldEQ(t, res34.ToString(), res34_cmp.ToString()) // got
-1157920892373161954235709850086879078532699846656405640394575840079131296399333, expected 3
        shouldEQ(t, res35.ToString(), res35_cmp.ToString()) // got -0, expected 0
        shouldEQ(t, res36.ToString(), res36_cmp.ToString()) // got -0, expected 0
```

```
        print("Done")
}

func shouldEQ(t *testing.T, got, expected interface{}) {
        if got != expected {
                t.Errorf("got %v, expected %v", got, expected)
        }
}
```

Recommendation

Since `i256` is a package related to arithmetic operations, it is recommended to modify the implementation to ensure consistent handling of the sign for zero values. This will help prevent potential issues arising from negative zero and maintain consistency across operations.

Fix Comment

Zero value checks are performed for the arithmetic operations Add, Sub, Quo, and Rem, and the signs are unified to false.

Fix commit hash: [0b3d12589d7125fbde660d372aee70844927f9b2]

```
        // Ensure zero is always positive
        if z.abs.IsZero() {
                z.neg = false
        }
```

18. When limitCaller is false, incorrect behavior of AllowCallFromOnly prevents function usage.

ID: Gnoswap-18 Severity: Medium
Type: Denial of Service Difficulty: High
File: common/allow_non_gnoswap_contracts.gno

Issue

The Pool Realm uses the AllowCallFromOnly function to restrict function calls to the Position Realm. This function is defined in the Common Realm and checks if PrevRealm matches the package received as an argument. However, since this function exists in the Common function, the value of PrevRealm returns the Pool Realm. Therefore, when using this function, the function call always fails.

```
func AllowCallFromOnly(allowPath string) {
        if !limitCaller {
                prevPath := std.PrevRealm().PkgPath()
                if prevPath != allowPath {
                        panic("caller is not allowed to call this function")
                }
        }
}
```

Additionally, in the Position Realm, the caller is restricted when limitCaller is true, but in the AllowCallFromOnly function used by the Pool Realm, the caller is restricted when limitCaller is false. This discrepancy between the two Realms can lead to function call restrictions.

```
        if common.GetLimitCaller() {
                // only user or staker can call
                isUserCalled := std.IsOriginCall()
                isStakerCalled := std.PrevRealm().Addr() == consts.STAKER_ADDR
                if !(isUserCalled || isStakerCalled) {
                        panic("POSITION] position.gno__Mint() || only user or staker can call")
                }
        }
```

Recommendation

We recommend using GetCallerAt instead of PrevRealm in the AllowCallFromOnly function. Also, we recommend changing the logic to ensure consistent behavior based on the true/false value of limitCaller.

Fix Comment

The logic has been updated to allow only the Limit Caller to invoke functions via limit_caller.gno, and this has been confirmed to apply to the Mint, Burn, Collect, and Swap functions.

Fix commit hash: [1766975d8e4f9df4453ce204c48a43d2bfb97110]

```
package common

import (
        "std"

        "gno.land/r/gnoswap/v2/consts"
)

var (
        limitCaller bool = true
)

func GetLimitCaller() bool {
        return limitCaller
}

func SetLimitCaller(v bool) {
        caller := std.GetOrigCaller()
        if caller != consts.GNOSWAP_ADMIN {
                panic("must be called by admin")
        }
        limitCaller = v
}
```

```
func Mint(
        token0Path string,
        token1Path string,
        fee uint32,
        recipient string,
        tickLower int32,
        tickUpper int32,
        _liquidityAmount string, // uint128
) (string, string) { // uint256 x2
        if common.GetLimitCaller() {
                prev := std.PrevRealm().PkgPath()
                if prev != consts.POSITION_PATH {
                        panic(ufmt.Sprintf("[POOL] pool.gno__Mint() || prev(%s) !=
consts.POSITION_PATH(%s)", prev, consts.POSITION_PATH))
                }
        }
...
```

```
func Burn(
        token0Path string,
        token1Path string,
        fee uint32,
        tickLower int32,
        tickUpper int32,
        _liquidityAmount string, // uint128
) (string, string) { // uint256 x2
```

```
        if common.GetLimitCaller() {
                prev := std.PrevRealm().PkgPath()
                if prev != consts.POSITION_PATH {
                        panic(ufmt.Sprintf("[POOL] pool.gno__Burn() || prev(%s) !=
consts.POSITION_PATH(%s)", prev, consts.POSITION_PATH))
                }
        }

...
```

```
func Collect(
        token0Path string,
        token1Path string,
        fee uint32,
        _recipient string,
        tickLower int32,
        tickUpper int32,
        _amount0Requested string, // uint128
        _amount1Requested string, // uint128
) (string, string) { // uint128 x2
        if common.GetLimitCaller() {
                prev := std.PrevRealm().PkgPath()
                if prev != consts.POSITION_PATH {
                        panic(ufmt.Sprintf("[POOL] pool.gno__Collect() || prev(%s) !=
consts.POSITION_PATH(%s)", prev, consts.POSITION_PATH))
                }
        }

...
```

```
func Swap(
        token0Path string,
        token1Path string,
        fee uint32,
        _recipient string,
        zeroForOne bool,
        _amountSpecified string, // int256
        _sqrtPriceLimitX96 string, // uint160
        _payer string, // router
) (string, string) { // int256 x2
        if common.GetLimitCaller() {
                prev := std.PrevRealm().PkgPath()
                if prev != consts.ROUTER_PATH {
                        panic(ufmt.Sprintf("[POOL] pool.gno__Swap() || prev(%s) !=
consts.ROUTER_PATH(%s)", prev, consts.ROUTER_PATH))
                }
        }
...
```

19. A single user can take all the Fees generated in a Pool.

ID: Gnoswap-19
Type: Input Validation
File: position/position.gno

Severity: Medium
Difficulty: Low

Issue

The Position Realm provides a CollectFee function that distributes Swap fees generated in the Pool to gnft holders. When the PositionKey is the same, users should receive Swap Fees proportional to the liquidity they provided. In the current implementation, the first user to call CollectFee when a Swap Fee occurs takes all the fees, thereby taking all the fees that other users should receive.

```
        ...
        positionKey := positionKeyCompute(GetOrigPkgAddr(), position.tickLower, position.tickUpper)
        pool := pl.GetPoolFromPoolPath(position.poolKey)
        _feeGrowthInside0LastX128, _feeGrowthInside1LastX128 :=
pool.PoolGetPositionFeeGrowthInside0LastX128(positionKey),
pool.PoolGetPositionFeeGrowthInside1LastX128(positionKey)
        feeGrowthInside0LastX128 := u256.MustFromDecimal(_feeGrowthInside0LastX128.ToString())
        feeGrowthInside1LastX128 := u256.MustFromDecimal(_feeGrowthInside1LastX128.ToString())

        position.feeGrowthInside0LastX128 = feeGrowthInside0LastX128
        position.feeGrowthInside1LastX128 = feeGrowthInside1LastX128

        // check user wugnot amount
        // need this value to unwrap fee
        userWugnot := wugnot.BalanceOf(a2u(std.GetOrigCaller()))

        amount0, amount1 := pl.Collect(
                token0,
                token1,
                fee,
                std.GetOrigCaller().String(),
                position.tickLower,
                position.tickUpper,
                consts.MAX_UINT64,
                consts.MAX_UINT64,
        )

        positions[tokenId] = position

        // handle withdrawal fee
        withoutFee0, withoutFee1 := pl.HandleWithdrawalFee(tokenId, token0, amount0, token1, amount1,
position.poolKey)
        ...
```

PoC

```
func TestCollectFeeWithTwoUser(t *testing.T) {
        BeforeEachTest(t)
        t.Run("mint and swap fee check in multiple user mint", func(t *testing.T) {
                std.TestSetRealm(gsaRealm)
                std.TestSetOrigCaller(gsa)

                bar.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
                baz.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
                bar.Approve(a2u(consts.ROUTER_ADDR), consts.UINT64_MAX)
                baz.Approve(a2u(consts.ROUTER_ADDR), consts.UINT64_MAX)

                pool := GetPool(barPath, bazPath, 3000)
                tickSpacing := pool.tickSpacing

                tokenId_res1, liquidity_res1, amount0_res1, amount1_res1:= pn.Mint(
                        barPath,// token0 string,
                        bazPath,// token1 string,
                        3000,   // fee uint32,
                        minTick,// tickLower int32,
                        maxTick,// tickUpper int32,
                        "10000000",      // _amount0Desired string, // *u256.Uint // 100e18
                        "10000000",      // _amount1Desired string, // *u256.Uint // 100e18
                        "0",    // _amount0Min string, // *u256.Uint
                        "0",    // _amount1Min string, // *u256.Uint
                        time.Now().Unix() + 1000,        // deadline int64,
                        user1Adderss.String(),   // mintTo string
                )

                tokenId_res2, liquidity_res2, amount0_res2, amount1_res2:= pn.Mint(
                        barPath,// token0 string,
                        bazPath,// token1 string,
                        3000,   // fee uint32,
                        minTick,// tickLower int32,
                        maxTick,// tickUpper int32,
                        "10000000",      // _amount0Desired string, // *u256.Uint // 100e18
                        "10000000",      // _amount1Desired string, // *u256.Uint // 100e18
                        "0",    // _amount0Min string, // *u256.Uint
                        "0",    // _amount1Min string, // *u256.Uint
                        time.Now().Unix() + 1000,        // deadline int64,
                        user2Adderss.String(),   // mintTo string
                )

                // ====== Swap to accrue fees ======
                pr.SwapRoute(
                        barPath, //inputToken string,
                        bazPath,  //outputToken string,
                        "10000000",//_amountSpecified string, // int256
                        "EXACT_IN", //swapType string,
                        barPath+":"+bazPath+":3000", //strRouteArr string, // []string
                        "100",//quoteArr string, // []int
                        "0",//_tokenAmountLimit string, // uint256
                )

                pr.SwapRoute(
                        bazPath, //inputToken string,
                        barPath,  //outputToken string,
                        "10000000",//_amountSpecified string, // int256
                        "EXACT_IN", //swapType string,
                        bazPath+":"+barPath+":3000", //strRouteArr string, // []string
                        "100",//quoteArr string, // []int
                        "0",//_tokenAmountLimit string, // uint256
```

```
                )

                // ====== Burn 0 to update fee ======
                positionRealm := std.NewUserRealm(consts.POSITION_ADDR)
                std.TestSetRealm(positionRealm)
                Burn(
                        barPath, // token0Path string,
                        bazPath, // token1Path string,
                        uint32(3000), // fee uint32,
                        minTick,// tickLower int32,
                        maxTick, // tickUpper int32,
                        "0",    // _liquidityAmount string, // uint128
                )

                // ====== Collect fees and copare ======
                // user1
                std.TestSetRealm(user1Realm)
                std.TestSetOrigCaller(user1Adderss)
                bar.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
                baz.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
                userBarBalanceBeforeCollect_1 := bar.BalanceOf(a2u(user1Adderss))
                userBazBalanceBeforeCollect_1 := baz.BalanceOf(a2u(user1Adderss))
                tokenId_res3, withoutFee0_res3, withoutFee1_res3, positionPoolKey_res3 :=
pn.CollectFee(tokenId_res1)
                userBarBalanceAfterCollect_1 := bar.BalanceOf(a2u(user1Adderss))
                userBazBalanceAfterCollect_1 := baz.BalanceOf(a2u(user1Adderss))
                println("user1 collect fee ", userBarBalanceAfterCollect_1 -
userBarBalanceBeforeCollect_1, userBazBalanceAfterCollect_1 - userBazBalanceBeforeCollect_1)

                // user2
                std.TestSetRealm(user2Realm)
                std.TestSetOrigCaller(user2Adderss)
                bar.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
                baz.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
                userBarBalanceBeforeCollect_2 := bar.BalanceOf(a2u(user2Adderss))
                userBazBalanceBeforeCollect_2 := baz.BalanceOf(a2u(user2Adderss))
                tokenId_res4, withoutFee0_res4, withoutFee1_res4, positionPoolKey_res4 :=
pn.CollectFee(tokenId_res2)
                userBarBalanceAfterCollect_2 := bar.BalanceOf(a2u(user2Adderss))
                userBazBalanceAfterCollect_2 := baz.BalanceOf(a2u(user2Adderss))
                println("user2 collect fee ", userBarBalanceAfterCollect_2 -
userBarBalanceBeforeCollect_2, userBazBalanceAfterCollect_2 - userBazBalanceBeforeCollect_2)
                shouldEQ(t, (userBarBalanceAfterCollect_1 - userBarBalanceBeforeCollect_1) -
(userBarBalanceAfterCollect_2 - userBarBalanceBeforeCollect_2), 0)
                shouldEQ(t, (userBazBalanceAfterCollect_1 - userBazBalanceBeforeCollect_1)-
(userBazBalanceAfterCollect_2 - userBazBalanceBeforeCollect_2), 0)
        })
        AfterEachTest(t)
}
```

Recommendation

We recommend changing the logic to allocate Swap Fees according to the user's Liquidity ratio, referencing the Uniswap V3 code.

Fix Comment

It has been verified that the modifications have been made to allocate fees proportionally to the Liquidity of the user's position.

Fix pr link: https://github.com/gnoswap-labs/gnoswap/pull/319

```
        tokensOwed0 := position.tokensOwed0
        tokensOwed1 := position.tokensOwed1

        {
                diff := new(u256.Uint).Sub(feeGrowthInside0LastX128, position.feeGrowthInside0LastX128)
                mulDiv := u256.MulDiv(diff, position.liquidity, u256.MustFromDecimal(consts.Q128))

                tokensOwed0 = new(u256.Uint).Add(tokensOwed0, mulDiv)
        }

        {
                diff := new(u256.Uint).Sub(feeGrowthInside1LastX128, position.feeGrowthInside1LastX128)
                mulDiv := u256.MulDiv(diff, position.liquidity, u256.MustFromDecimal(consts.Q128))

                tokensOwed1 = new(u256.Uint).Add(tokensOwed1, mulDiv)
        }
```

20. GNFT positions with duplicate TokenURIs can be created.

ID: Gnoswap-20
Type: Logic Error/Bug
File: gnft/gnft.gno

Severity: Low
Difficulty: Low

Issue

When creating a staker's position gnft, a unique random token URI is generated for each token ID. The random generator uses the block time as the seed value. If two or more tokens are created and URIs are generated in the same block, they will use the same block time. Therefore, multiple position gnfts with duplicate token URIs can be created

```
func SetTokenURI(tid grc721.TokenID) {
        // rand instance
        seed1 := uint64(time.Now().Unix())
        seed2 := uint64(time.Now().UnixNano())
        pcg := rand.NewPCG(seed1, seed2)
        r := rand.New(pcg)

        tokenURI := genImageURI(r)

        ok, err := gnft.SetTokenURI(tid, grc721.TokenURI(tokenURI))
        if !ok {
                panic(err.Error())
        }

        std.Emit(
                "GNOSWAP",
                "m_origCaller", std.GetOrigCaller().String(),
                "m_prevRealm", std.PrevRealm().PkgPath(),
                "p_tokenId", string(tid),
                "tokenURI", tokenURI,
                "SetTokenURI", "SetTokenURI",
        )
}
```

PoC

```
func TestMintPositionAndCheckURI(t *testing.T) {
        std.TestSetRealm(gsaRealm)
        bar.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
        foo.Approve(a2u(consts.POOL_ADDR), consts.UINT64_MAX)
        poolPath := "gno.land/r/onbloc/bar:gno.land/r/onbloc/foo:500"
        SetPoolTier(poolPath, 1)

        tokenId1, _, _, _, _ := MintAndStake(
                barPath,
                fooPath,
                fee500,
                8000,
```

```
                12000,
                "50000000",
                "50000000",
                "0",
                "0",
                max_timeout,
        )

        tokenId2, _, _, _, _ := MintAndStake(
                barPath,
                fooPath,
                fee500,
                9000,
                13000,
                "50000000",
                "50000000",
                "0",
                "0",
                max_timeout,
        )

        tid1:= grc721.TokenID(ufmt.Sprintf("%d", tokenId1))
        tid2:= grc721.TokenID(ufmt.Sprintf("%d", tokenId2))
        tokenURI1 := gnft.GetTokenURI(tid1)
        tokenURI2 := gnft.GetTokenURI(tid2)

        shouldEQ(t, tokenURI1 == tokenURI2, false)
}
```

Recommendation

It is recommended to use both block time and token ID as seeds when generating random values.

Fix Comment

The SetTokenURI function has been modified to use both block time and token ID as seeds, ensuring that duplicate URIs are not generated.

Fix commit hash: [1ccb5ee74d69c248df793ec1ac6a64f3c3149ef5]

```
func SetTokenURI(tid grc721.TokenID) {
        // rand instance
        seed1 := uint64(time.Now().Unix() + int64(TotalSupply()))
        seed2 := uint64(time.Now().UnixNano() + int64(TotalSupply()))
...
}
```

Test Cases

Range	Num	Spec	Test Function	Pass/Fail
Pool	1	fails if amount is 0	TestMint_fail_if_amount_is_0	Fail
	2	fails if tickLower greater than tickUpper	TestMint_fail_if_tickLower_greater_than_tickUpper	Pass
	3	fails if tickLower less than min tick	TestMint_fail_if_tickLower_less_than_min_tick	Pass
	4	fails if tickUpper greater than max tick	TestMint_fail_if_tickUpper_greater_than_max_tick	Pass
	5	fails if amount exceeds the max	TestMint_fail_if_amount_exceeds_the_max	Pass
	6	fails if total amount at tick exceeds the max	TestMint_fail_if_total_amount_at_tick_exceeds_the_max	Pass
	7	initial balances	TestSuccess_case_init_balance	Pass
	8	initial tick	TestSuccess_case_init_tick	Pass
	9	transfers token0 only	TestSuccess_case_transfer_token0_only	Pass
	10	max tick with max leverage	TestSuccess_case_max_tick_with_max_leverage	Pass
	11	works for max tick	TestSuccess_case_work_for_max_tick	Pass
	12	removing works	TestSuccess_case_removing_works	Pass
	13	adds liquidity to liquidityGross	TestSuccess_case_adds_liquidity_to_liquidityGross	Pass
	14	removes liquidity from liquidityGross	TestSuccess_case_removes_liquidity_from_liquidityGross	Pass

	15	clears tick lower if last position is removed	TestSuccess_case_clear_tick_lowers_if_last_poistion_is_removed	Pass
	16	clears tick upper if last position is removed	TestSuccess_case_clear_tick_lowers_if_last_poistion_is_removed2	Pass
	17	only clears the tick that is not used at all	TestSuccess_case_only_clears_the_tick_that_is_not_used_at_all	Pass
	18	price within range: transfers current price of both tokens	TestSuccess_price_within_range_transfers_current_price_of_both_tokens	Pass
	19	initializes lower tick	TestSuccess_initializes_lower_tick	Pass
	20	initializes upper tick	TestSuccess_initializes_upper_tick	Pass
	21	works for min/max tick	TestSuccess_case_works_for_min_max_tick	Pass
	22	transfers token1 only	TestSuccess_case_transfer_token1_only	Pass
	23	min tick with max leverage	TestSuccess_case_min_tick_with_max_leverage	Pass
	24	work for min tick	TestSuccess_case_work_for_min_tick	Pass
	25	fails if not initialized	TestCreateFooBarPool_Should_be_failed_if_price_is_too_low	Fail
	26	fails if starting price is too low	TestCreateFooBarPool_Should_be_failed_if_price_is_too_low	Pass
	27	fails if starting price is too high	TestCreateFooBarPool_Should_be_failed_if_price_is_too_high	Pass
	28	can be initialized at MIN_SQRT_RATIO	TestCreateFooBarPool_can_be_init_at_MIN_SQRT_RATIO	Fail

	29	can be initialized at MAX_SQRT_RATIO - 1	TestCreateFooBarPool_can_be_init_at_MAX_SQRT_RATIO_Sub1	Pass
	30	sets initial variables	TestCreateFooBarPool_set_initial_variables	Pass
	31	can create a pool	TestCreateFooBarPool	Pass
	32	can create multiple pools	TestCreateBarBazPool	Pass
	33	can mint liquidity	TestMintFooBarLiquidity	Pass
	34	can mint liquidity in different pools	TestMintBarBazLiquidity	Pass
	35	can get pools	TestApiGetPools	Pass
	36	can get withdrawal fee	TestGetWithdrawalFee	Pass
	37	cannot set withdrawal fee without permission	TestSetWithdrawalFeeNoPermission	Pass
	38	cannot set withdrawal fee out of range	TestSetWithdrawalFeeFeeOutOfRange	Pass
	39	can set withdrawal fee	TestSetWithdrawalFee	Pass
	40	can get pool creation fee	TestGetPoolCreationFee	Pass
	41	cannot set pool creation fee without permission	TestSetPoolCreationFeeNoPermission	Pass
	42	can set pool creation fee	TestSetPoolCreationFee	Pass
Router	1	fails if amount is 0	TestMint_fail_if_amount_is_0	Pass
	2	initial balances	TestMint_initial_balances	Pass
	3	initial tick	TestMint_initial_tick	Pass

	4	above current price transfers token0 only	TestMint_above_current_price_transfers_token0_only	Pass
	5	max tick with max leverage	TestMint_max_tick_with_max_leverage	Pass
	6	works for max tick	TestMint_works_for_max_tick	Pass
	7	removing works	TestMint_removing_works	Pass
	8	adds liquidity to liquidityGross	TestMint_adds_liquidity_to_liquidityGross	Pass
	9	removes liquidity from liquidityGross	TestMint_removes_liquidity_from_liquidityGross	Pass
	10	clears tick lower if last position is removed	TestMint_clears_tick_lower_if_last_position_is_removed	Pass
	11	clears tick upper if last position is removed	TestMint_clears_tick_upper_if_last_position_is_removed	Pass
	12	only clears the tick that is not used at all	TestMint_only_clears_the_tick_that_is_not_used_at_all	Pass
	13	does not write an observation	TestMint_does_not_write_an_observation	Pass
	14	transfers current price of both tokens	TestMint_transfers_current_price_of_both_tokens	Pass
	15	initializes lower tick	TestMint_initializes_lower_tick	Pass
	16	initializes upper tick	TestMint_initializes_upper_tick	Pass
	17	works for min/max tick	TestMint_works_for_min_max_tick	Pass
	18	removing works	TestMint_removing_works_2	Pass
	19	writes an observation	TestMint_writes_an_observation	Pass

	20	transfers token1 only	TestMint_transfers_token1_only	Pass
	21	min tick with max leverage	TestMint_min_tick_with_max_leverage	Pass
	22	works for min tick	TestMint_works_for_min_tick	Pass
	23	removing works	TestMint_removing_works_3	Pass
	24	does not write an observation	TestMint_does_not_write_an_observation_2	Pass
	25	protocol fees accumulate as expected during swap	TestMint_protocol_fees_accumulate_as_expected_during_swap	Pass
	26	positions are protected before protocol fee is turned on	TestMint_positions_are_protected_before_protocol_fee_is_turned_on	Pass
	27	poke is not allowed on uninitialized position	TestMint_poke_is_not_allowed_on_uninitialized_position	Pass
	28	current tick accumulator increases by tick over time	TestObserve_current_tick_accumulator_increases_by_tick_over_time	Pass
	29	current tick accumulator after single swap	TestObserve_current_tick_accumulator_after_single_swap	Pass
	30	current tick accumulator after two swaps	TestObserve_current_tick_accumulator_after_two_swaps	Pass
Liquidity Math	1	1 + 0	TestAddDelta_1	Pass
	2	1 + -1	TestAddDelta_2	Pass
	3	1 + 1	TestAddDelta_3	Pass
	4	2**128-15 + 15 overflows	TestAddDelta_4	Pass
	5	0 + -1 underflows	TestAddDelta_5	Pass

	6	3 + -4 underflows	TestAddDelta_6	Pass
SqrtPrice Math	1	fails if price is zero	TestGetNextSqrtPriceFromInput_1	Fail
	2	fails if liquidity is zero	TestGetNextSqrtPriceFromInput_2	Fail
	3	fails if input amount overflows the price	TestGetNextSqrtPriceFromInput_3	Fail
	4	any input amount cannot underflow the price	TestGetNextSqrtPriceFromInput_4	Pass
	5	returns input price if amount in is zero and zeroForOne = true	TestGetNextSqrtPriceFromInput_5	Pass
	6	returns input price if amount in is zero and zeroForOne = false	TestGetNextSqrtPriceFromInput_6	Pass
	7	returns the minimum price for max inputs	TestGetNextSqrtPriceFromInput_7	Pass
	8	input amount of 0.1 token1	TestGetNextSqrtPriceFromInput_8	Pass
	9	input amount of 0.1 token0	TestGetNextSqrtPriceFromInput_9	Pass
	10	amountIn > type(uint96).max and zeroForOne = true	TestGetNextSqrtPriceFromInput_10	Pass
	11	can return 1 with enough amountIn and zeroForOne = true	TestGetNextSqrtPriceFromInput_11	Pass
	12	fails if price is zero	TestGetNextSqrtPriceFromOutput_1	Pass
	13	fails if liquidity is zero	TestGetNextSqrtPriceFromOutput_2	Fail

	14	fails if output amount is exactly the virtual reserves of token0	TestGetNextSqrtPriceFromOutput_3	Pass
	15	fails if output amount is greater than virtual reserves of token0	TestGetNextSqrtPriceFromOutput_4	Pass
	16	fails if output amount is exactly the virtual reserves of token1	TestGetNextSqrtPriceFromOutput_5	Pass
	17	succeeds if output amount is just less than the virtual reserves of token1	TestGetNextSqrtPriceFromOutput_6	Pass
	18	puzzling echidna test	TestGetNextSqrtPriceFromOutput_7	Pass
	19	returns input price if amount in is zero and zeroForOne = true	TestGetNextSqrtPriceFromOutput_8	Pass
	20	returns input price if amount in is zero and zeroForOne = false	TestGetNextSqrtPriceFromOutput_9	Pass
	21	output amount of 0.1 token1	TestGetNextSqrtPriceFromOutput_10	Pass
	22	output amount of 0.1 token1	TestGetNextSqrtPriceFromOutput_11	Pass
	23	reverts if amountOut is impossible in zero for one direction	TestGetNextSqrtPriceFromOutput_12	Pass
	24	reverts if amountOut is impossible in one for zero direction	TestGetNextSqrtPriceFromOutput_13	Pass
	25	returns 0 if liquidity is 0	TestSqrtPriceMathGetAmount0DeltaStr_1	Pass

	26	returns 0 if prices are equal	TestSqrtPriceMathGetAmount0DeltaStr_2	Pass
	27	returns 0.1 amount1 for price of 1 to 1.21	TestSqrtPriceMathGetAmount0DeltaStr_3	Pass
	28	works for prices that overflow	TestSqrtPriceMathGetAmount0DeltaStr_4	Pass
	29	returns 0 if liquidity is 0	TestSqrtPriceMathGetAmount0DeltaHelper_1	Pass
	30	returns 0 if prices are equal	TestSqrtPriceMathGetAmount0DeltaHelper_2	Pass
	31	returns 0.1 amount1 for price of 1 to 1.21	TestSqrtPriceMathGetAmount0DeltaHelper_3	Pass
	32	the sub between the result of roundup and rounddown should be eq to 1	TestSqrtPriceMathGetAmount0DeltaHelper_4	Pass
	33	works for prices that overflow	TestSqrtPriceMathGetAmount0DeltaHelper_5	Pass
	34	returns 0 if liquidity is 0	TestSqrtPriceMathGetAmount1DeltaHelper_1	Fail
	35	returns 0 if prices are equal	TestSqrtPriceMathGetAmount1DeltaHelper_2	Pass
	36	returns 0.1 amount1 for price of 1 to 1.21	TestSqrtPriceMathGetAmount1DeltaHelper_3	Pass
	37	sqrtP * sqrtQ overflows	TestSwapComputation	Pass

SwapMath	1	exact amount in that gets capped at price target in one for zero	TestSwapMathComputeSwapStepStr_1	Pass
	2	exact amount out that gets capped at price target in one for zero	TestSwapMathComputeSwapStepStr_2	Pass
	3	exact amount in that is fully spent in one for zero	TestSwapMathComputeSwapStepStr_3	Pass
	4	amount out is capped at the desired amount out	TestSwapMathComputeSwapStepStr_4	Pass
	5	target price of 1 uses partial input amount	TestSwapMathComputeSwapStepStr_5	Pass
	6	entire input amount taken as fee	TestSwapMathComputeSwapStepStr_6	Pass
	7	handles intermediate insufficient liquidity in zero for one exact output case	TestSwapMathComputeSwapStepStr_7	Pass
	8	handles intermediate insufficient liquidity in one for zero exact output case	TestSwapMathComputeSwapStepStr_8	Pass
TickMath	1	throws for too low (MIN_TICK - 1)	TestTickMathGetSqrtRatioAtTick_1	Fail
	2	throws for too high (MAX_TICK + 1)	TestTickMathGetSqrtRatioAtTick_2	Fail
	3	min tick	TestTickMathGetSqrtRatioAtTick_3	Pass
	4	min tick +1	TestTickMathGetSqrtRatioAtTick_4	Pass
	5	max tick - 1	TestTickMathGetSqrtRatioAtTick_5	Pass

	6	min tick ratio is less than js implementation	TestTickMathGetSqrtRatioAtTick_6	Pass
	7	max tick ratio is greater than js implementation	TestTickMathGetSqrtRatioAtTick_7	Pass
	8	max tick	TestTickMathGetSqrtRatioAtTick_8	Pass
	9	various tick values (positive and negative)	TestTickMathGetSqrtRatioAtTick_Result	Pass
	10	MIN_SQRT_RATIO equals getSqrtRatioAtTick(MIN_TICK)	TestMIN_SQRT_RATIO	Pass
	11	MAX_SQRT_RATIO equals getSqrtRatioAtTick(MAX_TICK)	TestMAX_SQRT_RATIO	Pass
	12	getTickAtSqrtRatio throws for too low	TestTickMathGetTickAtSqrtRatio_1	Fail
	13	getTickAtSqrtRatio throws for too high	TestTickMathGetTickAtSqrtRatio_2	Fail
	14	getTickAtSqrtRatio for ratio of min tick	TestTickMathGetTickAtSqrtRatio_3	Pass
	15	getTickAtSqrtRatio for ratio of min tick + 1	TestTickMathGetTickAtSqrtRatio_4	Pass
	16	getTickAtSqrtRatio for ratio of max tick - 1	TestTickMathGetTickAtSqrtRatio_5	Pass
	17	getTickAtSqrtRatio for various ratios	TestTickMathGetTickAtSqrtRatio_Result	Fail
Tick	1	returns the correct value for low fee	TestTickTickSpacingToMaxLiquidityPerTick_1	Pass
	2	returns the correct value for medium fee	TestTickTickSpacingToMaxLiquidityPerTick_2	Pass

	3	returns the correct value for high fee	TestTickTickSpacingToMaxLiquidityPerTick_3	Pass
	4	returns the correct value for entire range	TestTickTickSpacingToMaxLiquidityPerTick_4	Pass
	5	returns the correct value for 2302	TestTickTickSpacingToMaxLiquidityPerTick_5	Pass
	6	returns all for two uninitialized ticks if tick is inside	TestTickGetFeeGrowthInside_1	Pass
	7	returns 0 for two uninitialized ticks if tick is above	TestTickGetFeeGrowthInside_2	Pass
	8	returns 0 for two uninitialized ticks if tick is below	TestTickGetFeeGrowthInside_3	Pass
	9	subtracts upper tick if below	TestTickGetFeeGrowthInside_4	Pass
	10	subtracts lower tick if above	TestTickGetFeeGrowthInside_5	Pass
	11	subtracts upper and lower tick if inside	TestTickGetFeeGrowthInside_6	Pass
	12	works correctly with overflow on inside tick	TestTickGetFeeGrowthInside_7	Pass
	13	flips from zero to nonzero	TestTickUpdate_1	Pass
	14	does not flip from nonzero to greater nonzero	TestTickUpdate_2	Pass
	15	flips from nonzero to zero	TestTickUpdate_3	Pass
	16	does not flip from nonzero to lesser nonzero	TestTickUpdate_4	Pass

	17	reverts if total liquidity gross is greater than max	TestTickUpdate_5	Pass
	18	nets the liquidity based on upper flag	TestTickUpdate_6	Pass
	19	reverts on overflow liquidity gross	TestTickUpdate_7	Pass
	20	assumes all growth happens below ticks lte current tick	TestTickUpdate_8	Pass
	21	does not set any growth fields if tick is already initialized	TestTickUpdate_9	Pass
	22	does not set any growth fields for ticks gt current tick	TestTickUpdate_10	Pass
	23	deletes all the data in the tick	TestClear_1	Pass
	24	flips the growth variables	TestTickCross_1	Pass
	25	two flips are no op	TestTickCross_2	Pass
FullMath	1	reverts if denominator is 0	TestMulDiv_1	Pass
	2	reverts if denominator is 0 and numerator overflows	TestMulDiv_2	Pass
	3	reverts if output overflows uint256	TestMulDiv_3	Pass
	4	reverts on overflow with all max inputs	TestMulDiv_4	Pass
	5	all max inputs	TestMulDiv_5	Pass

	6	accurate without phantom overflow	TestMulDiv_6 and TestMulDiv_6_1	Pass
	7	accurate with phantom overflow	TestMulDiv_7	Pass
	8	accurate with phantom overflow and repeating decimal	TestMulDiv_8	Pass
	9	reverts if denominator is 0 (MulDivRoundingUp)	TestMulDivRoundingUp_1	Pass
	10	reverts if denominator is 0 and numerator overflows (MulDivRoundingUp)	TestMulDivRoundingUp_2	Pass
	11	reverts if output overflows uint256 (MulDivRoundingUp)	TestMulDivRoundingUp_3	Pass
	12	reverts on overflow with all max inputs (MulDivRoundingUp)	TestMulDivRoundingUp_4	Pass
	13	reverts if mulDiv overflows 256 bits after rounding up	TestMulDivRoundingUp_5	Pass
	14	reverts if mulDiv overflows 256 bits after rounding up case 2	TestMulDivRoundingUp_6	Pass
	15	all max inputs (MulDivRoundingUp)	TestMulDivRoundingUp_7	Pass
	16	accurate without phantom overflow (MulDivRoundingUp)	TestMulDivRoundingUp_8	Pass

	17	accurate with phantom overflow (MulDivRoundingUp)	TestMulDivRoundingUp_9	Pass
	18	accurate with phantom overflow and repeating decimal (MulDivRoundingUp)	TestMulDivRoundingUp_10	Pass
TickBitmap	1	is false at first	TestTickInit_1	Pass
	2	is flipped by #flipTick	TestTickInit_1	Pass
	3	is flipped back by #flipTick	TestTickInit_2	Pass
	4	is not changed by another flip to a different tick	TestTickInit_3	Pass
	5	is not changed by another flip to a different tick on another word	TestTickInit_4	Pass
	6	flips only the specified tick	TestTickFlip_1	Pass
	7	reverts only itself	TestTickFlip_2	Pass
	8	returns tick to right if at initialized tick	TestTicknextInitializedTickWithinOneWord_1	Pass
	9	returns tick to right if at initialized tick	TestTicknextInitializedTickWithinOneWord_2	Pass
	10	returns the tick directly to the right	TestTicknextInitializedTickWithinOneWord_3	Pass
	11	returns the tick directly to the right	TestTicknextInitializedTickWithinOneWord_4	Pass
	12	skips half word	TestTicknextInitializedTickWithinOneWord_5	Pass

	13	skips half word	TestTicknextInitializedTickWithinOneWord_6	Pass
	14	returns same tick if initialized	TestTickLteEqTrue_1	Pass
	15	returns tick directly to the left of input tick if not initialized	TestTickLteEqTrue_2	Pass
	16	will not exceed the word boundary	TestTickLteEqTrue_3	Pass
	17	at the word boundary	TestTickLteEqTrue_4	Pass
	18	word boundary less 1 (next initialized tick in next word)	TestTickLteEqTrue_5	Pass
	19	entire empty word	TestTickLteEqTrue_6	Pass
	20	boundary is initialized	TestTickLteEqTrue_7	Pass
Others	1	-	TestStakeAndGetBack	Fail
	2	-	TestMintAndStakeAndGetBack	Fail
	3	-	TestCreateExternalIncentive	Fail
	4		TestCollectFeeWithTwoUser	Fail
	5	-	TestInt256MinusZero	Fail
	6	-	TestCreateWUGNOTWUGNOTPool	Fail
	7	-	TestMintPositionSwapFeeCheck	Fail
	8	-	TestCheckPosition	Fail
	9	-	TestSwapRouteWugnotquxExactIn	Fail
	10		TestMintPositionAndCheckUniquURI	Fail

DISCLAIMER

This report does not provide investment advice, guarantee the suitability of the business model, or ensure that the code is free of bugs and secure. It is intended solely for the discussion of known technical issues. In addition to the issues outlined in the report, there may be undiscovered problems, such as defects on the mainnet. To ensure secure code, it is essential to address the identified issues and conduct thorough testing.

Appendix. A

Severity Level

CRITICAL	Must be addressed as a vulnerability that has the potential to seize or freeze substantial sums of money.
HIGH	Has to be fixed since it has the potential to deny users compensation or momentarily freeze assets.
MEDIUM	Vulnerabilities that could halt services, such as DoS and Out-of-Gas, need to be addressed.
LOW	Issues that do not comply with standards or return incorrect values
TIPS	Tips that makes the code more usable or efficient when modified

Difficulty Level

	Low	Medium	High
Privilege	anyone	Miner/Block Proposer	Admin/Owner
Capital needed	Small or none	Gas fee or volatile as price change	More than exploited amount
Probability	100%	Depend on environment	Hard as mining difficulty

Vulnerability Category

Arithmetic	• Integer under/overflow vulnerability • floating point and rounding accuracy
Access & Privilege Control	• Manager functions for emergency handle • Crucial function and data access • Count of calling important task, contract state change, intentional task delay
Denial of Service	• Unexpected revert handling • Gas limit excess due to unpredictable implementation
Miner Manipulation	• Dependency on the block number or timestamp. • Frontrunning
Reentrancy	•Proper use of Check-Effect-Interact pattern. •Prevention of state change after external call • Error handling and logging.
Low-level Call	• Code injection using delegatecall • Inappropriate use of assembly code
Off-standard	• Deviate from standards that can be an obstacle of interoperability.
Input Validation	• Lack of validation on inputs.
Logic Error/Bug	• Unintended execution leads to error.
Documentation	•Coherency between the documented spec and implementation
Visibility	• Variable and function visibility setting
Incorrect Interface	• Contract interface is properly implemented on code.

End of Document