Nemo

Audit Report

contact@bitslab.xyz

https://twitter.com/movebit_

Mon Dec 30 2024

Nemo Audit Report

1 Executive Summary

1.1 Project Information

Description	Nemo Protocol is a decentralized finance application.
Type	DeFi
Auditors	MoveBit
Timeline	Thu Nov 07 2024 - Mon Dec 30 2024
Languages	Move
Platform	Sui
Methods	Architecture Review, Unit Testing, Manual Review
Source Code	https://github.com/nemo-protocol/nemo
Commits	1efcf5aef3e9eaf70de56f6e5e727c4937d50119 62e73571b8cbdd363871f57473ae846a4e37d258 77b2640af5c9f30583dec6b7b39778d99e31551f fa7fae52b59733a526ca15faa4af2c59b4f0a4c4

1.2 Files in Scope

The following are the SHA1 hashes of the original reviewed files.

ID	File	SHA-1 Hash
FAC	nemo/sources/market/factory.move	051e7cb4161646384fc779d2fb7fcf0c1a2b6d3c
MMA12	nemo/sources/market/market_math.move	da65dff435489a8ab455f42c9e72bfb3eaa79b86
MGL	nemo/sources/market/market_global.move	06cb0217f0f3c8a789632afb1f6fd82ae080ae00
MPO	nemo/sources/market/market_position.move	4b799aff8c5283a7be9ba84ea0e47789a76c6c42
MAR1	nemo/sources/market/market.move	e3b4a06bfe7dc95d47074ebb992e48cfa67a5990
SY	nemo/sources/sy.move	bf6c3f6d73e34fca5f73e3a736721c9a7b372343
YFA1	nemo/sources/py/yield_factory.move	4e843a33496ada6ecc6fa9ba769441951220260e
PY	nemo/sources/py/py.move	532f24cf1f978e42802f96d5e610819990a2bc55
PPO	nemo/sources/py/py_position.move	e0d9a65572d97ad95e370af28eb71033a718c340
ORA	nemo/sources/oracle.move	d49cb97ee3bb44c24a2954112f9035a7fcd364d0

1.3 Issue Statistic

Item	Count	Fixed	Partially Fixed	Acknowledged
Total	30	25	1	4
Informational	5	3	0	2
Minor	8	5	1	2
Medium	4	4	0	0
Major	8	8	0	0
Critical	5	5	0	0

1.4 MoveBit Audit Breakdown

MoveBit aims to assess repositories for security-related issues, code quality, and compliance with specifications and best practices. Possible issues our team looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Integer overflow/underflow by bit operations
- Number of rounding errors
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting
- Unchecked CALL Return Values
- The flow of capability
- Witness Type

1.5 Methodology

The security team adopted the **"Testing and Automated Analysis"**, **"Code Review"** and **"Formal Verification"** strategy to perform a complete security test on the code in a way that is closest to the real attack. The main entrance and scope of security testing are stated in the conventions in the "Audit Objective", which can expand to contexts beyond the scope according to the actual testing needs. The main types of this security audit include:

(1) Testing and Automated Analysis

Items to check: state consistency / failure rollback / unit testing / value overflows / parameter verification / unhandled errors / boundary checking / coding specifications.

(2) Code Review

The code scope is illustrated in section 1.2.

(3) Formal Verification(Optional)

Perform formal verification for key functions with the Move Prover.

(4) Audit Process

- Carry out relevant security tests on the testnet or the mainnet;
- If there are any questions during the audit process, communicate with the code owner in time. The code owners should actively cooperate (this might include providing the latest stable source code, relevant deployment scripts or methods, transaction signature scripts, exchange docking schemes, etc.);
- The necessary information during the audit process will be well documented for both the audit team and the code owner in a timely manner.

2 Summary

This report has been commissioned by Nemo Protocol to identify any potential issues and vulnerabilities in the source code of the Nemo smart contract, as well as any contract dependencies that were not part of an officially recognized library. In this audit, we have utilized various techniques, including manual code review and static analysis, to identify potential vulnerabilities and security issues.

During the audit, we identified 30 issues of varying severity, listed below.

ID	Title	Severity	Status
PY-1	Incorrect Reward Update Method In `update_user_interest` Function	Critical	Fixed
PY-2	Mismatch Between Function Name And Implementation In `borrow_pt_amount`	Critical	Fixed
PY-3	Price Cache Time is Too Long	Critical	Fixed
PY-4	`get_price()` Function Precision Limit	Informational	Fixed
SY-1	`FlashLoan` Has Unnecessary `store` Capabilities	Medium	Fixed
SY-2	Invalid Slippage Check	Minor	Fixed
ACL-1	`remove_role` Doesn't Check For The Existence Of Permissions	Major	Fixed
FAC-1	Potential DoS Vulnerability In `create_new_market_with_raw_values` Function	Major	Fixed
FAC-2	Potential Dos of `create_py()`	Major	Fixed

FAC-3	The Market Creation Time May be Too Small	Minor	Fixed
FP6-1	Inaccurate Calculation in `truncate_up()`	Major	Fixed
MAR-1	`get_rate_anchor` Status Update Error	Major	Fixed
MAR-2	The Initial Liquidity Provider Will Lose Some LP	Minor	Acknowledged
MAR-3	`swap_exact_pt_for_sy()` Function Lacks Expiration Check	Minor	Fixed
MAR-4	Initial Liquidity Ratio Manipulation Lacks Slippage Control	Minor	Partially Fixed
MAR-5	Computational Optimization	Informational	Fixed
MMA-1	Unreachable Instance of `market_exchange_rate_below_one` Error	Informational	Fixed
MPO-1	Lacks Authentication	Critical	Fixed
ORA-1	Instantaneous Price Dependence	Medium	Fixed
ORA-2	Epoch Issues	Minor	Fixed
YFA-1	SyInterestPostExpiry Calculation Error	Critical	Fixed
YFA-2	Lack of Permission Validation for `init_config()`	Major	Fixed
YFA-3	Inconsistency Between `PyState` and `PyPosition`	Major	Fixed

YFA-4	Lack of Version Check	Medium	Fixed
YFA-5	Expiration Time Boundary Value Check Error	Minor	Fixed
MAR1-1	LP Slippage Check Issues	Major	Fixed
MAR1-2	`add_liquidity_single_sy` Lack of Market CAP checks	Medium	Fixed
MAR1-3	Code Optimization	Minor	Acknowledged
MAR1-4	`get_market_state` Does Not Determine Whether The Market Is Empty	Informational	Acknowledged
MAR1-5	`reward_rate` Field Design Issues	Informational	Acknowledged

3 Participant Process

Here are the relevant actors with their respective abilities within the Nemo Smart Contract : The client replied that:

1. All rights of the contract are managed by the foundation.
2. Only users can deposit and withdraw money under any conditions, and the foundation has no right to do so.

Owner

- The owner can call the `create_new_market_with_raw_values` function to create a new market.
- The owner can call the `register_underlying_token` to create `underlying_token`.
- The owner can call the `update_config` function to update the `MarketFactoryConfig` info.
- The owner can use the `add_role` function to add a role to a member.

User

- User can wrap the interest-bearing token into `SYCoin` through `sy`, which can be divided into PT/YT, participate in Market transactions, and get back the principal through redemption.
- User needs to get the latest price through the oracle contract when using functions in the market.
- User can add or remove liquidity in the market contract.
- User can use the swap function in the market contract to exchange SY/PT/YT.

4 Findings

PY-1 Incorrect Reward Update Method In `update_user_interest` Function

Severity: Critical

Status: Fixed

Code Location:

nemo/sources/py/py.move#415

Descriptions:

In the `update_user_interest` function, the user's reward is being overwritten using the `set_accured` method instead of updating it incrementally with a cumulative method. This leads to a loss of previously accrued rewards, as the new value completely replaces the existing one.

Suggestion:

It is recommended to replace the call to `set_accured(accured)` with a method that properly accumulates the user's rewards.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

PY-2 Mismatch Between Function Name And Implementation In `borrow_pt_amount`

Severity: Critical

Status: Fixed

Code Location:

nemo/sources/py/py.move#110

Descriptions:

The function `borrow_pt_amount` implies its purpose is to borrow an amount of PT (Principal Token). However, the implementation incorrectly uses the `mint_py` function with the first parameter set to amount, which represents the YT (Yield Token) value, not the PT value. This results in the function minting YT instead of borrowing PT, leading to a discrepancy between the function's name, its intended behavior, and the actual implementation.

Suggestion:

It is recommended that the code be changed to the correct logic.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

PY-3 Price Cache Time is Too Long

Severity: Critical

Status: Fixed

Code Location:

nemo/sources/py/py.move#304

Descriptions:

When `do_cache_index_same_block` is true, when using cached prices, the corresponding update time period is epoch. The epoch time in sui is usually about 24 hours. In the pendle project, `block.number` is used, which is usually about 15 seconds. If the cached price is too long and `new_index` is not used, serious errors in price calculation will occur.

```
public fun current_py_index<SYCoin: drop>(
        do_cache_index_same_block: bool,
        exchange_rate: FixedPoint64,
        py_state: &mut PyState<SYCoin>,
        ctx: &mut TxContext,
    ): FixedPoint64 {
        if (do_cache_index_same_block && py_state.py_index_last_updated_epoch ==
ctx.epoch()) {
            py_state.py_index_stored
```

Suggestion:

It is recommended to use cached prices for shorter time periods.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

PY-4 `get_price()` Function Precision Limit

Severity: Informational

Status: Fixed

Code Location:

nemo/sources/py/py.move

Descriptions:

The `get_price()` function extracts the price information from PriceVoucher, but in the `calc_scoin_to_coin()` function, the price calculation relies on a fixed unit (unit = 1e9), which means that only tokens that meet the expected precision can calculate the price correctly. For example, when the token precision is 1e18, the price of deflationary tokens and transfer tokens may be calculated incorrectly.

Suggestion:

It is recommended to use only the intended kind of tokens.

Resolution:

The client avoids this risk through manual control.

SY-1 `FlashLoan` Has Unnecessary `store` Capabilities

Severity: Medium

Status: Fixed

Code Location:

nemo/sources/sy.move#37

Descriptions:

Structure `FlashLoan` is used as a receipt for `sy` loaned `SYCoin` and can only be destroyed in the `repay` function. This is consistent with the `Hot-Potato` design pattern, so `FlashLoan` does not need a `store` capability to ensure that it must be destroyed within a transaction.

Suggestion:

It is recommended that unneeded capabilities be removed.

SY-2 Invalid Slippage Check

Severity: Minor

Status: Fixed

Code Location:

nemo/sources/sy.move

Descriptions:

The `deposit` function and the `redeem` function are used to deposit `YieldCoin` type tokens into the system and return the corresponding `SYCoin` type voucher. The `min_amount_out` parameter is used to ensure that the `SYCoin` obtained by the user after depositing the token is not less than the expected minimum value. This parameter is provided by the user, not dynamically calculated according to market conditions, so it cannot effectively prevent slippage.

```
assert!(
    share >= min_amount_out,
    error::sy_insufficient_sharesOut()
);
```

Suggestion:

Sy tokens may use exchangerate to actually execute how many tokens are needed to mint. When packaging, it is necessary to check based on the actual number. It is recommended to check the slippage as the actual number of tokens.

ACL-1 `remove_role` Doesn't Check For The Existence Of Permissions

Severity: Major

Status: Fixed

Code Location:

lib/sources/acl.move#116

Descriptions:

The `remove_role` function removes a user's permissions without checking whether the role of the user exists or not, and removing it directly may cause problems. For example, if the user's permissions are currently `1<<11`, and admin removes a non-existing permission such as 1<<5, the result is `1<<11-1<<5` according to the code, which results in the user having all permissions from 6-10.

```
>>> bin((1<<11))
'0b100000000000'
>>> bin((1<<5))
'0b100000'
>>> bin((1<<11)-(1<<5))
'0b11111100000'
```

Suggestion:

It is recommended to determine if the role exists when removing it.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

FAC-1 Potential DoS Vulnerability In `create_new_market_with_raw_values` Function

Severity: Major

Status: Fixed

Code Location:

nemo/sources/market/factory.move

Descriptions:

A function can only be called by a user with `create_market_role` permission, then the function will call `config.add(market_id);` to add the `market_id` to the `MarketFactoryConfig`. But since `market_global::add` is a public function and `MarketFactoryConfig` is a share object, it means that any user can directly call add function to add market_id to `MarketFactoryConfig` which may cause the bag in `MarketFactoryConfig` to be too large.

Suggestion:

It is recommended to change the permissions of the `add` function.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

FAC-2 Potential Dos of `create_py()`

Severity: Major

Status: Fixed

Code Location:

nemo/sources/market/factory.move#32

Descriptions:

The lack of permission checking for `yield_factory::create` can lead to DOS problems where a user may call the `create_py` function to set a malicious `expiry` parameter, resulting in normal users no longer being able to create.

Suggestion:

It's recommended to take measures to avoid this issue.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

FAC-3 The Market Creation Time May be Too Small

Severity: Minor

Status: Fixed

Code Location:

nemo/sources/market/factory.move#93

Descriptions:

When creating a market, the creator checks that the `expiry` is greater than the current time, and does not check the minimum cycle time, which may result in the creation of an `expiry` that is too small.

```
assert!(
    expiry > clock.timestamp_ms(),
    error::market_pt_expired()
);
```

Suggestion:

It is recommended to modify the code to:

```
expiry > clock.timestamp_ms() + confing.minTime
```

FP6-1 Inaccurate Calculation in `truncate_up()`

Severity: Major

Status: Fixed

Code Location:

nemo_math/sources/fixed_point64.move#70

Descriptions:

In the `truncate_up()`, the `truncated_val` results from a 64-bit right shift. If it is shifted right by another 64 bits, the result will be 0. The correct way to round up should be to perform a 64-bit left shift and compare it with the original value.

```
public fun truncate_up(val: FixedPoint64): u64 {
    let truncated_val = (val.value >> 64) as u64;
    if(((truncated_val as u256) >> 64) < (val.value as u256)) {
        truncated_val + 1
    } else {
        truncated_val
    }
}
```

Suggestion:

It is recommended to update `((truncated_val as u256) >> 64` to `((truncated_val as u256) << 64`.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MAR-1 `get_rate_anchor` Status Update Error

Severity: Major

Status: Fixed

Code Location:

nemo/sources/market/market.move#777

Descriptions:

In the non-initialized state, when Total is not equal to 0, `rate_anchor` should be updated. In the else branch of the `mint_lp_internal` function, when `market.lp_supply != 0`, the current function state should be set to `false` to update the scalar.

```
if(!init) {
    rate_anchor = market_math::get_rate_anchor(
```

```
let exchange_rate = get_exchange_rate(
factory_config,
py_state,
market,
clock,
fixed_point64::create_from_raw_value(price_rate),
true, //bug
ctx
);
```

Suggestion:

It is recommended to modify the status(false) of the two `get_exchange_rate()` functions in the else branch to update `rate_anchor`.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MAR-2 The Initial Liquidity Provider Will Lose Some LP

Severity: Minor

Status: Acknowledged

Code Location:

nemo/sources/market/market.move#177

Descriptions:

In the `mint_lp_internal()` function, when `market.lp_supply` is 0, the system subtracts a constant `MINIMUM_LIQUIDITY` from the liquidity calculation. This means that the actual amount of liquidity tokens a user receives when providing initial liquidity will be reduced by `MINIMUM_LIQUIDITY`. This reduction will not be returned to the user, but will remain in the market.

On the other hand, `lp_to_user` may be calculated as 0, especially when liquidity is equal to `MINIMUM_LIQUIDITY`. This means that users may not receive any liquidity tokens when providing liquidity.

```
if(market.lp_supply == 0) {
    let liquidity =  math128::sqrt((pt_amount_mut as u128) * (sy_value as u128)) as u64;
    assert!(liquidity >= MINIMUM_LIQUIDITY, error::market_liquidity_too_low());
    let lp_to_user=  liquidity - MINIMUM_LIQUIDITY;
    let lp_reserve = MINIMUM_LIQUIDITY;
    let sy_balance = sy_amount_mut.split(sy_value);
    // deduct pt from user
    py::split_pt(
        py_position,
        pt_amount_mut
    );
    // add pt to market
    market.total_pt = market.total_pt + pt_amount_mut;
    // add sy to market
    market.total_sy.join(sy_balance);
    // add lp to market supply
    market.lp_supply = lp_to_user + lp_reserve;
    // add lp to user
    market_position.set_lp_amount(lp_to_user);
```

Suggestion:

It is recommended that when calculating lp_to_user, ensure that the result is always greater than 0, and return the overcharged amount to the user after the time expires.

MAR-3 `swap_exact_pt_for_sy()` Function Lacks Expiration Check

Severity: Minor

Status: Fixed

Code Location:

nemo/sources/market/market.move#578

Descriptions:

The `swap_exact_pt_for_sy()` function calls `swap_exact_pt_for_sy_internal` without checking the expiration time. The execution `let time_to_expire = market.expiry - clock::timestamp_ms(clock);` may have exceeded `market.expiry`, resulting in execution failure.

Suggestion:

It is recommended to add expiration time check in `swap_exact_yt_for_sy()` and `swap_exact_pt_for_sy()`, as well as `swap_sy_for_exact_pt()`.

MAR-4 Initial Liquidity Ratio Manipulation Lacks Slippage Control

Severity: Minor

Status: Partially Fixed

Code Location:

nemo/sources/market/market.move

Descriptions:

The initial liquidity ratio manipulation in the `mint_lp()` function lacks slippage control. It does not check the "minOut" amount, which means that when the robot monitors the initial liquidity addition from the memory pool, it can run ahead, resulting in an unexpected liquidity ratio added by the user, causing an unexpected `exchangeRate`.

```
public fun mint_lp<SYCoin: drop>(
    version: &Version,
    sy_coin: Coin<SYCoin>,
    pt_amount: u64,
    price_voucher: oracle::PriceVouch
    py_position: &mut py_position::Py
    py_state: &mut py::PyState<SYCoin
    factory_config: &YieldFactoryConf
    market: &mut MarketState<SYCoin>,
    clock: &clock::Clock,
    ctx: &mut tx_context::TxContext
): (Coin<SYCoin>, MarketPosition) {
version::assert_current_version(version);
let (py_balance, _) = py::get_py_amount(py_position);
assert!(clock::timestamp_ms(clock) < py_position.expiry(), error::market_expired());
assert!(pt_amount > 0, error::market_pt_amount_is_zero());
assert!(py_balance >= pt_amount, error::market_insufficient_pt_in_for_mint_lp());
assert!(py_position.py_state_id() == market.py_state_id, error::market_invalid_py_state());
...
assert!(balance::value(&sy_amount) > 0, error::market_sy_amount_is_zero());
...
assert!(liquidity >= MINIMUM_LIQUIDITY, error::market_liquidity_too_low());
...
```

```
assert!(!market.last_ln_implied_rate.equal(fixed_point64::zero()),
error::market_ln_implied_rate_is_zero());
```

In the current implementation, the initial liquidity provider can add any amount of PT/SY (as long as last_ln_implied_rate > 0), the next liquidity provider must provide at the same rate. This allows an attacker to manipulate the exchange rate by adding liquidity with a large amount of PT and a small amount of SY.

```
market.last_ln_implied_rate = get_ln_implied_rate(exchange_rate, time_to_expire);
assert!(!market.last_ln_implied_rate.equal(fixed_point64::zero()),
error::market_ln_implied_rate_is_zero());
```

example

`scalarRoot` = 1e9

`initialAnchor` = 1.1e9

`timeExporiy` = 1 year

user call `mint_lp()` **as:**

syDesired = 10000e9

ptDesired = 10000e9

minLpOut = 0

attacker front-run user'tx call `mint_lp()` **as:**

syDesired = 2000

ptDesired = 50000

minLpOut = 0

Market State:

totalsy = 2000

totalpt = 50000

totallp = 2000

after this user'tx call mint:

netlppt = ptDesired * market.totalLp / market.totalPt = 400e9

netlpsy = syDesired * market.totalLp / market.totalsy = 10000e9

Market State:

totalsy = 400e9+2000

totalpt = 10000e9 + 50000

totallp = 400e9+2000

After this step, the exchange rate and implied rate both change by dozens of times due to the pt/sy ratio.

At this point, the attacker can take advantage of the manipulated exchange rate, back-run the user'tx, and make a profit.

Suggestion:

It is recommended to check the PT/SY equivalent when trading for the first time.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MAR-5 Computational Optimization

Severity: Informational

Status: Fixed

Code Location:

nemo/sources/market/market.move#196

Descriptions:

Note that the total amount of `p_to_user + lp_reserve` calculated in `market.lp_supply` is exactly equal to `liquidity`. We can use `liquidity` directly to reduce the contract calculation amount, optimize the code and save gas.

```
market.lp_supply = lp_to_user + lp_reserve;
```

Suggestion:

It is recommended to use `liquidity` to replace the calculation.

MMA-1 Unreachable Instance of `market_exchange_rate_below_one` Error

Severity: Informational

Status: Fixed

Code Location:

nemo/sources/market/market_math.move#28

Descriptions:

The `market_exchange_rate_below_one` error code will never be triggered, since `exchange_rate` is calculated by `get_exchange_rate_from_implied_rate`, and the values by the exponential function are all greater than `ONE` so this will not happen.

Suggestion:

It is recommended that meaningless code be removed based on protocol design considerations.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MPO-1 Lacks Authentication

Severity: Critical

Status: Fixed

Code Location:

nemo/sources/market/market_position.move#104

Descriptions:

The reference function in the `market_position` contract is public. It does not check any permissions to add or set `market_position.lp_amount`. As long as the user uses `mint_lp` to obtain the `MarketPosition` object, it can be modified directly.

```
public fun set_lp_amount(
    market_position: &mut MarketPosition,
    lp_amount: u64
) {
    market_position.lp_amount = lp_amount;
}
public fun increase_lp_amount(
    market_position: &mut MarketPosition,
    lp_amount: u64
) {
    market_position.lp_amount = market_position.lp_amount + lp_amount;
}
public fun decrease_lp_amount(
    market_position: &mut MarketPosition,
    lp_amount: u64
) {
    market_position.lp_amount = market_position.lp_amount - lp_amount;
}
```

On the other hand, `PyPosition` can also modify `accured` directly through the set function.

```
public fun set_accured(
    py_position: &mut PyPosition,
    accured: FixedPoint64
) {
```

```
    py_position.accured = accured
}
```

Also, the `update_current_exchange_rate` function does not have any permission control, resulting in anyone being able to modify `current_exchange_rate`. The lack of permission checking for `yield_factory::create` can lead to DOS problems where a user may call the `create_py` function to set and malicious expiry parameter, resulting in normal users no longer being able to create.

Suggestion:

It is recommended to make sure that this is designed in accordance with the protocol and to add permission checks to the function.

ORA-1 Instantaneous Price Dependence

Severity: Medium

Status: Fixed

Code Location:

nemo/sources/oracle.move

Descriptions:

In the `calc_scoin_to_coin()` function, the price calculation depends on the instantaneous price calculation of `protocol::market`, which includes the stock, debt, cash, and revenue of the token. This dependency may lead to the following problems:

Users can use the instantaneous high price of the market to obtain a PriceVoucher when calling the get_price_voucher_from_x_oracle function. Since the price is calculated based on the current state of the market, users can artificially increase the price by manipulating the market state (for example, increasing the stock of tokens or reducing debt in the short term).

Users can use the high-priced PriceVouchers they obtain to mint more liquidity tokens (LP) for arbitrage.

```
public fun get_price_voucher_from_x_oracle<SYCoin: drop, UnderlyingToken: drop>(
    version: &protocol::version::Version,
    market: &mut protocol::market::Market,
    sy_state: &sy::State,
    clock: &Clock,
    ctx: &TxContext
): PriceVoucher<SYCoin> {
    assert!(
        sy::is_sy_bind_with_underlying_token<UnderlyingToken, SYCoin>(sy_state),
        error::sy_not_supported()
    );
    let unit = 1000000000;
    let coin_output = calc_scoin_to_coin(
        version,
        market,
        type_name::get<UnderlyingToken>(),
        clock,
```

```
        unit
    );
    let rate = fixed_point64::create_from_rational(coin_output as u128, unit as u128);

    PriceVoucher<SYCoin> {
        underlying_price: rate.get_raw_value(),
        epoch: ctx.epoch()
    }
}
```

On the other hand, when the user calls the `get_price_voucher_from_x_oracle()` function, we assume that the vendor code is updated once every 1000 seconds. The user calls and executes at the 999th second and the 1001st second, which will rely on two different prices for calculation. The root cause is that the price used by the user is updated when calling, while the price used by the contract is automatically obtained periodically, and they are not synchronized in process.

Suggestion:

It is recommended to confirm that the problem affects the price calculation dependency of the protocol market and restrict users from obtaining it multiple times in a short period of time or take other steps to mitigate the problem.

ORA-2 Epoch Issues

Severity: Minor

Status: Fixed

Code Location:

nemo/sources/oracle.move

Descriptions:

The `get_price` function is called to check if the epoch in `PriceVoucher` is equal to the epoch in the current `ctx`, but since `PriceVoucher` is a `hot-potato` type, `get_price_voucher_from_x_oracle` and `get_price` must be executed in the same transaction, so checking the epoch seem to work for nothing.

Suggestion:

It is recommended to ensure that this is as designed.

YFA-1 SyInterestPostExpiry Calculation Error

Severity: Critical

Status: Fixed

Code Location:

nemo/sources/py/yield_factory.move#369

Descriptions:

When calculating `sy_interest_post_expiry`, if `expired=true`, you need to subtract the user's value after converting the asset to get the excess funds before sending them to `factory_config.treasury`. The current judgment lacks the difference calculation, which leads to an error when all conversions are sent to `factory_config.treasury`.

```
let mut sy_interest_post_expiry = fixed_point64::zero();
if (expired) {
    sy_interest_post_expiry = sy::asset_to_sy(
        py::first_py_index(py_state),
        fixed_point64::from_uint64(amount_to_redeem)
    );
};
py::transfer_sy(
    factory_config.treasury,
    py_state,
    sy_interest_post_expiry,
    ctx
);
```

Suggestion:

It is recommended to modify the code to `syInterestPostExpiry = sy_interest_post_expiry - sy_amount_to_user`.

YFA-2 Lack of Permission Validation for `init_config()`

Severity: Major

Status: Fixed

Code Location:

nemo/sources/py/yield_factory.move#86

Descriptions:

The method `init_config()` lacks proper permissions, which means anyone can call it and brings potential risk.

Suggestion:

It is recommended to add permission to fix this issue.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

YFA-3 Inconsistency Between `PyState` and `PyPosition`

Severity: Major

Status: Fixed

Code Location:

nemo/sources/py/yield_factory.move#208

Descriptions:

The `mint_py_internal()` function does not verify the consistency between `PyState` and `PyPosition`. If inconsistent objects are used, it may prevent users from properly burning the `sy`.

Suggestion:

It is recommended to check the `state_id` is the same in the `mint_py_internal()` function.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

YFA-4 Lack of Version Check

Severity: Medium

Status: Fixed

Code Location:

nemo/sources/py/yield_factory.move

Descriptions:

In the `redeem_py_internal` and `redeem_due_interest` function, version checking is missing. This could allow operations between different versions to be compatible, potentially leading to unknown security risks.

Suggestion:

It is recommended to add a version check for the function.

YFA-5 Expiration Time Boundary Value Check Error

Severity: Minor

Status: Fixed

Code Location:

nemo/sources/py/yield_factory.move#313;

nemo/sources/py/yield_factory.move#412

Descriptions:

The boundary value check of the expired status does not match. When the current time reaches `expiry`, it is also included in the expiration time.

```
let expired = clock::timestamp_ms(clock) > py_position.expiry();
```

Suggestion:

It is recommended to modify the check to `clock::timestamp_ms(clock) >= py_position.expiry();`.

MAR1-1 LP Slippage Check Issues

Severity: Major

Status: Fixed

Code Location:

nemo/vendor/protocol/sources/market/market.move#136-184

Descriptions:

In the function `seed_liquidity`, when checking the LP slippage issue, the comparison should be based on `market_position.lp_amount > min_lp_amount` instead of using `market.lp_supply`. Additionally, this method might fail under certain conditions. If it is intended to be used only for the first addition of liquidity, a restriction such as `market.lp_supply == 0` should be enforced.

In the `mint_lp` method, the validation for `min_lp_amount` should ensure that `market_position.lp_amount >= min_lp_amount`.

Suggestion:

It's recommended to use `market_position.lp_amount > min_lp_amount` instead of `market.lp_supply` for slippage checks in `seed_liquidity`. Additionally, enforce `market.lp_supply == 0` for initial liquidity.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MAR1-2 `add_liquidity_single_sy` Lack of Market CAP checks

Severity: Medium

Status: Fixed

Code Location:

nemo/vendor/protocol/sources/market/market.move#811

Descriptions:

The function `add_liquidity_single_sy` lacks a check for Market CAP, leading to the possibility of adding liquidity in excess of CAP via the `add_liquidity_single_sy` function.

Suggestion:

When adding liquidity, it is recommended to judge whether the cap exceeds the market Cap.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

MAR1-3 Code Optimization

Severity: Minor

Status: Acknowledged

Code Location:

nemo/vendor/protocol/sources/market/market.move#136,184

Descriptions:

`seedLiquidity` and `mint_lp` methods, can move the judgment condition `assert!(market.market_cap == 0 || balance::value(&market.total_sy) <= market.market_cap, error::market_cap_exceeded());` to the `mint_lp_internal` function.

Suggestion:

It is recommended to optimize the code.

MAR1-4 `get_market_state` Does Not Determine Whether The Market Is Empty

Severity: Informational

Status: Acknowledged

Code Location:

nemo/vendor/protocol/sources/market/market.move#1115

Descriptions:

`get_market_state` does not determine `market.total_pt > 0` `total_asset > 0` the return value of `get_market_state` may have caused unexpected consequences.

Suggestion:

It is recommended that function `get_market_state` determine if market.total_pt, `total_asset` is zero.

MAR1-5 `reward_rate` Field Design Issues

Severity: Informational

Status: Acknowledged

Code Location:

nemo/vendor/protocol/sources/market/market.move

Descriptions:

The `reward_rate` field exists in the `YieldFactoryConfig` object but is not used anywhere else, make sure this is compliant with the protocol design. Also, the interest rate model in the protocol uses something like `(cash + debt - revenue)/market_coin_supply`, which in the early stages of the market if there is a reward, could cause inflation due to the `exchange_rate` calculation being attacked by the first minting (similar to the compound first minting bug).

Suggestion:

It is recommended to ensure that the reward_rate is as designed for the protocol.

Resolution:

The client has deleted this variable.

Appendix 1

Issue Level

- **Informational** issues are often recommendations to improve the style of the code or to optimize code that does not affect the overall functionality.
- **Minor** issues are general suggestions relevant to best practices and readability. They don't post any direct risk. Developers are encouraged to fix them.
- **Medium** issues are non-exploitable problems and not security vulnerabilities. They should be fixed unless there is a specific reason not to.
- **Major** issues are security vulnerabilities. They put a portion of users' sensitive information at risk, and often are not directly exploitable. All major issues should be fixed.
- **Critical** issues are directly exploitable security vulnerabilities. They put users' sensitive information at risk. All critical issues should be fixed.

Issue Status

- **Fixed:** The issue has been resolved.
- **Partially Fixed:** The issue has been partially resolved.
- **Acknowledged:** The issue has been acknowledged by the code owner, and the code owner confirms it's as designed, and decides to keep it.

Appendix 2

Disclaimer

This report is based on the scope of materials and documents provided, with a limited review at the time provided. Results may not be complete and do not include all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your own risk. A report does not imply an endorsement of any particular project or team, nor does it guarantee its security. These reports should not be relied upon in any way by any third party, including for the purpose of making any decision to buy or sell products, services, or any other assets. TO THE FULLEST EXTENT PERMITTED BY LAW, WE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, IN CONNECTION WITH THIS REPORT, ITS CONTENT, RELATED SERVICES AND PRODUCTS, AND YOUR USE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NOT INFRINGEMENT.