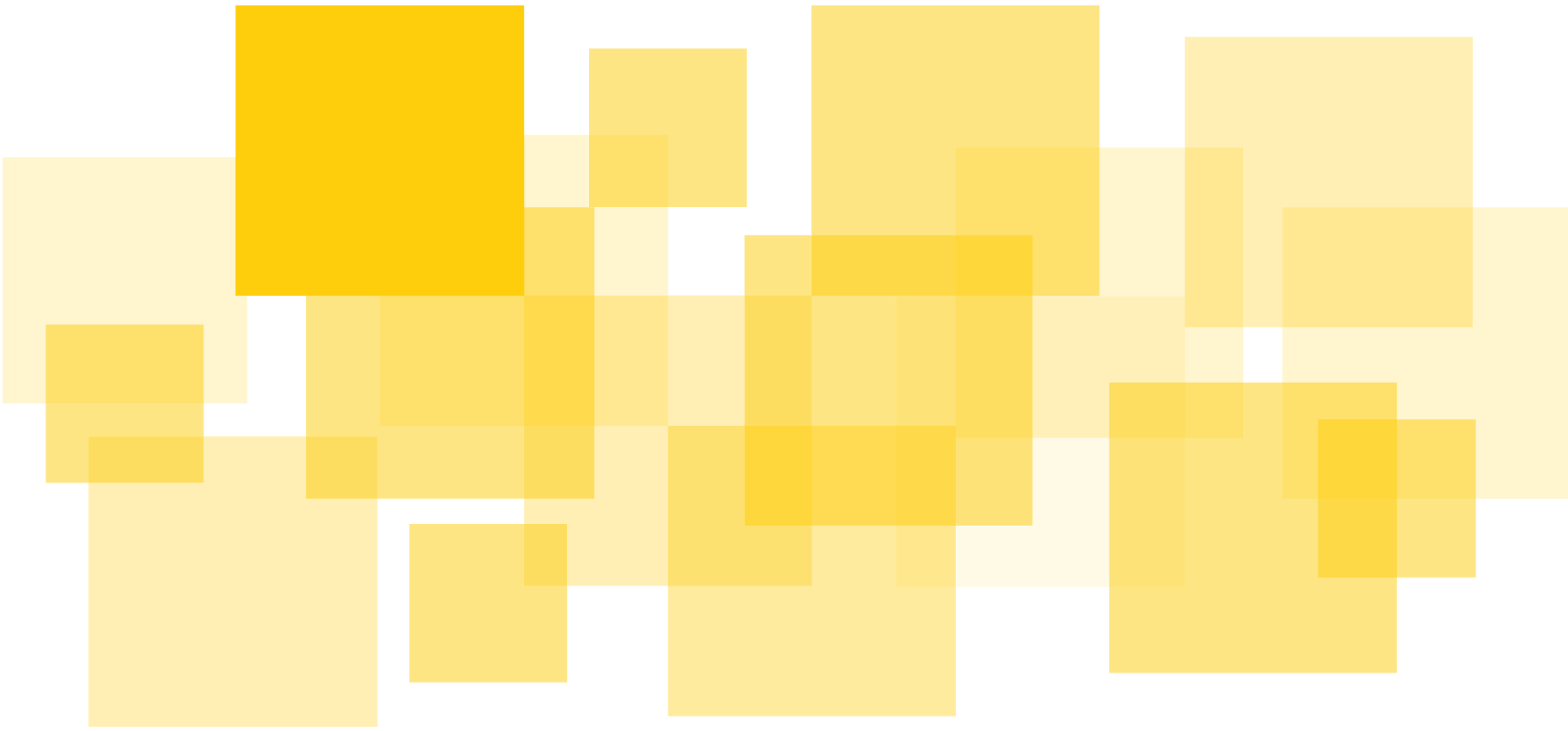
Security Audit Report

Element Finance Contract

Delivered: April 26th, 2021



Prepared for Element Finance by

runtime verification

Summary

Runtime Verification, Inc. conducted a security audit on the Element Finance smart contract. The audit was conducted by Daejun Park from March 15, 2021 to April 19, 2021.

Overall, several issues have been identified, including potential vulnerabilities to (flash loan based) external price manipulation and reentrancy attacks (A01, A05), implementation flaws (A02, A04, A06, A09), fixed-point arithmetics and rounding errors (A07, A08), and fault tolerance concerns (A03). A number of additional suggestions have also been made, regarding best practices (B02, B03, B08, B09, B10, B12, B14, B15), sanity checks (B04, B17, B18, B19), code readability (B01, B05, B06, B07, B13), and usability (B11, B16).

The code is well written and thoughtfully designed, following best practices.

Scope

The target of the audit is the smart contract source files at git-commit-id 637c6f9 as well as further code changes made up to b9ab609.

The audit focused on the following core contracts, reviewing their functional correctness and integration with external contracts:

- ConvergentCurvePool.sol
- InterestToken.sol
- Tranche.sol
- UserProxy.sol
- WrappedPosition.sol
- YVaultAssetProxy.sol
- factories/ConvergentPoolFactory.sol
- factories/InterestTokenFactory.sol
- factories/TrancheFactory.sol

The library contracts (`libraries/`) were given only lightweight review, and in particular, `DateString._daysToDate()` was assumed to be correct. The external contracts (`balancer-core-v2/` and `yearn-vaults/`) were also trusted to be correct and reliable.

The audit is limited in scope within the boundary of the Solidity contract only. Off-chain and client-side portions of the codebase as well as deployment and upgrade scripts are *not* in the scope of this engagement.

Assumptions

The audit is based on the following assumptions and trust model.

- The external balancer-core-v2 and yearn-vaults contracts are assumed to be correct, reliable, and secure.
- External token contracts conform to the ERC20 standard, especially they must revert in failures. Also, they do *not* allow any callbacks (e.g., ERC777) or any external contract calls.
- The external contracts associated with core contracts are not upgraded during the lifetime of each deployed core contract instance.
- For the authorizable contracts (UserProxy and ConvergentPoolFactory), the owners and authorized users are trusted to behave correctly and honestly.
- The governance account is trusted to behave honestly.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have followed the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code is vulnerable to known security issues and attack vectors. Third, we symbolically executed part of the compiled bytecode to systematically search for unexpected, possibly exploitable, behaviors at the bytecode level, that are due to EVM quirks or Solidity compiler bugs. Finally, we employed Firefly to measure the test coverage at the bytecode level, identifying missing test scenarios, and helping to improve the quality of tests.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Findings

A01: Vulnerability to external price manipulation

[Severity: High | Difficulty: High | Category: Security]

The Tranche contract logic (especially the principal token minting logic and the interest token redemption logic) largely relies on the assumption that the share price of the associated vault will monotonically increase throughout the lockup period. The logic currently employs only minimal protection measures for the case of the share price going down or fluctuating over the period. As such, the current behavior may be vulnerable to sophisticated (instant) price manipulation attacks, especially utilizing large amounts of flash loans.

Scenario

1. Alice picks a Tranche *X* that is currently locking up a large amount of deposits, say *N* Dai, (probably one that is near at the end of the lockup period).
2. Alice deposits a very large amount of underlying tokens (say 2*N* Dai for simplicity) to *X*.
3. Immediately after the end of the lockup period, (suppose that no one has yet withdrawn from *X*), Alice initiates a price manipulation attack for the vault associated with *X*. In particular, she first makes the share price (Dai/yDai) significantly drop (say dropped by 33% for simplicity) by selling a lot of major assets that the vault has invested.
4. Then, she withdraws all of her principal tokens. Note that she can do so without any loss because *X*'s underlying value is still sufficient to redeem her principal tokens. However, since the share price dropped by 33%, her withdrawal will cause *X*'s underlying value to be almost zero, which means all other depositors to *X* will be forced to lose most of their funds. Note also that Alice's withdrawal can force the vault to sell the assets, further dropping the vault-investing-asset price.
5. Now, Alice buys back a lot of vault-investing-assets at a steep discount, making a profit (due to the price difference from that of step (3)). Note that steps from (3) to (5) will be executed in a single transaction.

Discussion

Due to the zero sum game nature, it needs to be careful when realizing losses of users. The recent series of flash-loan-based attacks share a notable common pattern where attackers utilize losses of benign users in one DeFi system to make a profit for attackers in another DeFi system. Thus, it is much needed to employ certain protective measures that can help distinguish between legitimate losses due to market failures and "artificial" losses due to malicious price manipulation (e.g., by leveraging large flash loans). An example of such measures is to realize

losses (or gains, depending on the context) only after sufficient time has passed, which will make flash-loan-based price manipulation attacks much harder.

Recommendation

At the very least, fix the principal withdrawal logic to be more fair when realizing losses. That is, when "`position.balanceOfUnderlying(this)`" is significantly smaller than `valueSupplied`, the withdrawal can be temporarily disabled, or the redemption amount should be reduced proportional to the total amount of losses.

It can be also considered to improve the interest withdrawal logic to be able to suspend the withdrawal during the period of significant losses, or at least notify users and let them choose whether to wait until the recovery of temporary losses or to immediately withdraw regardless of losses.

For the long term, employ an additional protective measure that can adjust the (principal and interest) token redemption price over time, to be less vulnerable to instant-price-manipulation attacks.

Status

The client reported that PR 183 fixed this issue, but its impact on the other parts of the system have not been comprehensively reviewed.

A02: Flaws in minting principal tokens

[Severity: Low | Difficulty: Medium | Category: Functional Correctness]

When users deposit funds in the middle of a lockup period, they are minted less amount of principal tokens to deduct interests already accrued so far. However, the interest deduction logic has a flaw, leading to scenarios where users lose their funds even if the share price constantly goes up.

Scenario

1. Assume that there is no reserve in YVaultAssetProxy. Ignore any fees and rounding errors for simplicity.
2. Consider a 3 month lockup BTC vault, say from January 1st to April 1st. Let Q1, Q2, Q3, and Q4 be the price per share (i.e., BTC/yBTC) on January 1st, February 1st, March 1st, and April 1st, respectively.
3. Suppose Q1 = 1, Q2 = 1.5, Q3 = 1.8, and Q4 = 1.9.
4. Suppose Alice deposits 100 BTC on January 1st, Bob deposits 100 BTC on February 1st, and Charlie deposits 100 BTC on March 1st.
5. Then, Alice gets 100 fytBTC, Bob gets 50 fytBTC, and Charlie gets 0 fytBTC. They all get 100 ycyBTC.
6. On April 1st, 100 ycyBTC is redeemed by 90.74 BTC.
7. Thus, the total return for Alice, Bob, and Charlie is 190.74 BTC, 140.74 BTC, and 90.74 BTC, respectively. Now, Charlie complains because he lost even if the share price went up after he deposited (i.e., 1.8 -> 1.9).
8. Note that if they invested directly in the BTC vault, then their return would be 190 BTC, 126.67 BTC, 105.56 BTC, respectively. That means, Bob earned much more than the ideal.

Recommendation

Fix the interest deduction logic to prevent such a scenario.

Status

Fixed in 68595720616e4a74cf3a96e335ac265476ebcca6.

A03: Minting principal tokens with negative interest

[Severity: Low | Difficulty: High | Category: Fault Tolerance]

The principal token minting logic relies on the assumption that the vault will never return negative interest. The current logic leads to questionable behaviors (e.g., see the scenario below) when the assumption is not met. Since the negative interest can still happen in catastrophic failures of the vault and/or malicious price manipulation attempts, it is recommended to improve the logic to be more robust to deal with such cases.

Scenario

1. Assume that there is no reserve in YVaultAssetProxy. Ignore any fees and rounding errors for simplicity.
2. Consider a 3 month lockup BTC vault, say from January 1st to April 1st. Let Q1, Q2, Q3, and Q4 be the price per share (i.e., BTC/yBTC) on January 1st, February 1st, March 1st, and April 1st, respectively.
3. Suppose Q1 = 1, Q2 = 0.8, Q3 = 0.6, and Q4 = 1.5.
4. Suppose Alice deposits 100 BTC on January 1st, Bob deposits 100 BTC on February 1st, and Charlie deposits 100 BTC on March 1st.
5. Then, all of them equally get 100 fytBTC and 100 ycyBTC.
6. On April 1st, 100 ycyBTC is redeemed by 95.83 BTC.
7. Thus, the total return for Alice, Bob, and Charlie is equally 195.83 BTC. Then, Charlie complains because he got only ~95% gain even if the share price increased 2.5x after he deposited (i.e., 0.6 -> 1.5).
8. Note that if they invested directly in the BTC vault, then their return would be 150 BTC, 187.5 BTC, 250 BTC, respectively. That means, part of Charlie's interest was given to Alice.

Recommendation

For the short term, it can be considered to disable user deposits during the period of the vault yielding negative interest. For example, have "`require(holdingsValue >= valueSupplied)`" in prefundedDeposit().

For the long term, improve the logic to behave properly even in the case of the share price going down or fluctuating.

Status

Fixed in PR 183.

A04: Flaws in yearn vault integration

[Severity: High | Difficulty: Low | Category: Security / Functional Correctness]

Flaws in the yearn vault integration can be exploited to drain all the reserve funds.

Scenario

1. Suppose the initial reserve (100 Dai, 100 yDai). Suppose vault.report() is just executed, where the pricePerShare() value will increase from 1.0 Dai/yDai to 1.2 Dai/yDai over the next 6 hours. Immediately, Alice executes all the following steps from (2) to (6) in a single transaction.
2. Alice sells 100 Dai and receives 100 yDai. At this point, the reserve is (200 Dai, 0 yDai).
3. Now, Alice sells 1 Dai (or a tiny amount of Dai), and receives 0.83 yDai. At this point, the reserve is (0 Dai, 166.67 yDai). Note that this step triggers the actual vault.deposit() call.
4. Then, Alice sells 166.67 Dai, and receives 166.67 yDai. At this point, the reserve is (166.67 Dai, 0 yDai).
5. Alice sells 1 Dai (or a tiny amount of Dai), and receives 0.83 yDai. At this point, the reserve is (0 Dai, 138.89 yDai).
6. Alice repeats this until the profit is more than the gas cost. Say she repeats 10 times more, then the reserve becomes (0 Dai, 22.43 yDai).
7. Finally, after 6 hours, Alice can sell all the yDai tokens she has bought at discount. Her profit will amount to the loss of the reserve, which is ~180 USD in this case. In general, the loss of reserve could be almost 100% (depending on the gas cost).

Recommendation

Use the spot price (vault.totalAssets() / vault.totalSupply()) when minting shares for deposits.

Status

Fixed in [6c5c0e8fa57a539fc29d57d0a91557265485a219](6c5c0e8fa57a539fc29d57d0a91557265485a219).

A05: Reentrancy vulnerability

[Severity: High | Difficulty: High | Category: Security]

Some tokens (e.g., ERC777) allow callbacks from senders or receivers during token transfers. If the Element Finance system admits a token contract that allows the sender's callback to be executed after the token transfer, or the receiver's callback to be executed before the token transfer,[1] then the following exploit scenario exists.

Scenario

1. Suppose token *T* is a base token on the Element Finance system. Suppose that *T* allows the sender's hook to be executed after token transfer for some good reason.
2. Alice calls YVaultAssetProxy.reserveDeposit() with 100 *T* tokens.
3. In reserveDeposit(), it first transfers the 100 *T* tokens from Alice to the position contract.
4. Before returning back to reserveDeposit(), *T*.transferFrom() executes Alice's hook *H*.
5. In *H*, Alice calls WrappedPosition.prefundedDeposit(), which will mint the position share for 100 *T* tokens to Alice.
6. Then, *H* returns to *T*.transferFrom(), which in turn returns to reserveDeposit(). Then, reserveDeposit() resumes and mints the reserve share for 100 *T* tokens to Alice.
7. Now, Alice earns both the position and reverse shares. She can make 100% profit by selling or withdrawing both shares.

Note that if a token contract allows such a flexible callback, there exist other exploits similar to the above.

Recommendation

For the short term, review carefully when supporting new tokens to make sure they do not allow any callbacks that can exploit this vulnerability.

For the long term, consider employing more conservative reentrancy protection mechanisms such as ReentrancyGuard at extra gas cost.

Status

Acknowledged by the client.

[1] ERC777 does not allow this type of callback, but there could exist such token contracts in the future.

A06: Flaws in updating reserves

[Severity: High | Difficulty: Low | Category: Functional Correctness]

There are flaws in the reserves updating logic in the _deposit() and _withdraw() functions of YVaultAssetProxy. Specifically, in the following update, the new reserve of shares should have been "`localShares + shares - neededShares`" instead of "`shares - neededShares`":

```
        _setReserves(0, shares - neededShares);
```

YVaultAssetProxy.sol#L138

Also, in the following update, the new reserve of underlyings should have been "`localUnderlying + amountReceived - needed`" instead of "`amountReceived - needed`":

```
        _setReserves(amountReceived - needed, 0);
```

YVaultAssetProxy.sol#L176

Status

Fixed in 8fc1246536ca7b87e3df6e481a8c34a341077941.

A07: Fixed-point arithmetic errors in ConvergentCurvePool

[Severity: High | Difficulty: Low | Category: Functional Correctness]

The fixed-point arithmetics in ConvergentCurvePool are broken when the number of decimals of either the underlying or bond token is *not* 18. It blindly uses the 18-decimal-fixed-point arithmetic even if the operands are not of 18 decimals.

Status

Fixed in d4223f4acfdcfd38005d53d925ddc8ad764dceca.

A08: Undesirable failures for joinPool()/exitPool() due to rounding errors

[Severity: Low | Difficulty: Medium | Category: Functional Correctness]

There exist sufficiently many legitimate cases that do not pass the following checks:

```
// Safe math sanity checks
require(
     localFeeUnderlying >= (feesUsedUnderlying).div(percentFeeGov),
     "Underflow"
);
require(localFeeBond >= (feesUsedBond).div(percentFeeGov), "Underflow");
```

ConvergentCurvePool.sol#L606-L611

Analysis

Given `percentFeeGov` << 1, there exist many pairs of `localFeeUnderlying` and `feesUsedUnderlying` such that:

`localFeeUnderlying.mul(percentFeeGov) >= feesUsedUnderlying`

but:

`localFeeUnderlying < feesUsedUnderlying.div(percentFeeGov)`

This is because `mul()` is rounded half up instead of rounded down, and the rounded-up noise is amplified due to the division by a small fraction `percentFeeGov`.

Recommendation

Use the round-down multiplication and division, which always satisfies:

`localFeeUnderlying < localFeeUnderlying.mulDown(percentFeeGov).divDown(percentFeeGov)`

Status

Fixed in d4223f4acfdcfd38005d53d925ddc8ad764dceca and d15d07e2b1cf3ea09bd8bdb4fbb060ff874c7f86.

A09: Flaws in fee deduction logic

[Severity: Low | Difficulty: Medium | Category: Functional Correctness]

Flaws in the fee deduction logic lead to charging more fees than needed for certain cases.

Scenario

Case I: percentFeeGov is zero:

1. Suppose protocolSwapFee is 10% for simplicity.
2. Suppose feesUnderlying is 100 Dai, and feesBond is 100 fyDai at the moment.
3. Alice joins the pool, and 10 Dai and 10 fyDai are deducted to the vault as the protocol fee.
4. Immediately, without any trade made between, Bob joins the tool. At this point, no protocol fee should be deducted, but still 10 Dai and 10 fyDai will be deducted to the vault.
5. This can keep happening, and eventually all the fees could be drained to the vault, instead of going to the liquidity providers.

Case II: percentFeeGov is non-zero:

1. Suppose percentFeeGov is 10% and protocolSwapFee is 10% for simplicity.
2. Suppose the current balance of the pool is (1000 Dai, 1000 fyDai) for simplicity.[2]
3. Suppose feesUnderlying is 100 Dai, and feesBond is 50 fyDai at the moment. (The two fees could be significantly unbalanced. See below.)
4. Alice joins the pool, and the liquidity tokens that correspond to (5 Dai, 5 fyDai) will be minted to the governance account, and feesUnderlying will be updated to 50 Dai, and feesBond will be set to 0 fyDai (assuming no rounding errors). Also, (10 Dai, 5 fyDai) will be deducted to the vault as the protocol fee.
5. Immediately, Bob joins the pool, and no liquidity tokens are minted (because feesBond = 0), but still 5 Dai will be deducted to the vault, which is double-charged.
6. This can keep happening, and the bigger the unbalance ratio, the bigger the amount double-charged.

Note that the unbalanced ratio of the two trading fees is feasible even if there have been only symmetry tradings made between underlying and bond tokens. For example, suppose that most users somehow follow the following strategy:

- when they buy Dai and sell fyDai, they use onSwapGivenOut()
- when they sell Dai and buy fyDai, they use onSwapGivenIn()

[2] In normal operating conditions, the fyDai reserve should be bigger than the Dai reserve.

Then, feesBond could be much bigger than feesUnderlying. (In the extreme case, feesUnderlying could be zero.)

Recommendation

Fix the fee deduction logic.

Status

Fixed in d4223f4acfdcfd38005d53d925ddc8ad764dceca and d15d07e2b1cf3ea09bd8bdb4fbb060ff874c7f86.

Informative Findings

B01: Typographical errors

[Severity: Informative | Difficulty: N/A | Category: Code Refactoring]

For the following code in Tranche.constructor(), "`underlying = wpContract.token();`" can be simplified to "`underlying = localUnderlying;`" to save gas.

```
        IERC20 localUnderlying = wpContract.token();
        underlying = wpContract.token();
```

Tranche.sol#L50-L51

Status

Fixed in 069ad749bc953366928deeac67f8da9c21e140a5.

B02: SafeCast

[Severity: Informative | Difficulty: N/A | Category: Fault Tolerance]

In ConvergentCurvePool, Tranche, and YVaultAssetProxy, the explicit type conversion from `uint256` to `uint128` is used in several places. Since the conversion to a smaller-sized type does not check the range of the given value but silently truncates the higher-order bits, it is a better practice to explicitly check the value range (e.g., as in SafeCast) for each conversion. Note that even if values are expected to be always within the safe range in normal operating conditions, there could still exist certain catastrophic failures (possibly due to hidden bugs) that cause values to exceed the range. In such cases, having the explicit check will help to early detect and prevent any catastrophic failures from propagating further.

Status

Acknowledged by the client.

B03: SafeMath for ConvergentCurvePool

[Severity: Informative | Difficulty: N/A | Category: Fault Tolerance]

ConvergentCurvePool is to be compiled by Solidity 0.7.0 that does not have the builtin arithmetic overflow check. It is a better practice to systematically use SafeMath for every arithmetic operation, even if no overflow should occur in normal operating conditions, because it is hard to completely rule out the possibility of certain catastrophic failures (possibly due to hidden bugs) that could cause arithmetic overflow.

Status

Acknowledged by the client.

B04: Sanity check for onJoinPool() and onExitPool()

[Severity: Informative | Difficulty: N/A | Category: Fault Tolerance]

In onJoinPool() and onExitPool(), it is recommended to have a sanity check that the first argument is equal to `_poolId`. This ensures that the onJoinPool() hook is correctly routed by the vault, and can avoid minting LP tokens without ever receiving liquidity due to potential errors caused by the vault.

Status

Fixed in PR 184.

B05: Inconsistent boolean parameters

[Severity: Informative | Difficulty: N/A | Category: Code Refactoring]

Both solveTradeInvariant() and _assignTradeFee() handle the two trading modes (GivenIn vs GivenOut) based on their last boolean parameter. However, the boolean parameter of solveTradeInvariant() means the opposite of that of _assignTradeFee(), which is unintuitive and reduces code readability. It is recommended to make them consistent for maintenance in the future.

Status

Acknowledged by the client.

B06: Unused constants

[Severity: Informative | Difficulty: N/A | Category: Code Refactoring]

The `EPSILON` constant in ConvergentCurvePool is no longer used and can be removed for readability.

```
    // This is an error factor allowed in some fixed point operations
    // Equivalent to 10^-6 ie 0.0001% in 18 point fixed.
    uint256 public constant EPSILON = 1e12;
```

ConvergentCurvePool.sol#L50-L52

Status

Fixed in PR 184.

B07: Code simplification

[Severity: Informative | Difficulty: N/A | Category: Code Refactoring]

The following code can be further simplified using baseIndex and bondIndex:

```
IERC20[] memory tokens = new IERC20[](2);
if (_underlying < _bond) {
      tokens[0] = _underlying;
      tokens[1] = _bond;
} else {
      tokens[0] = _bond;
      tokens[1] = _underlying;
}
```

ConvergentCurvePool.sol#L100-L107

Status

Acknowledged by the client.

B08: Immutable

[Severity: Informative | Difficulty: N/A | Category: Code Refactoring]

The `_interestTokenFactory` storage variable can be declared as `immutable`:

```
    IInterestTokenFactory internal _interestTokenFactory;
```

TrancheFactory.sol#L24

Status

Fixed in PR 184.

B09: Signature malleability

[Severity: Informative | Difficulty: N/A | Category: Security]

While ERC20Permit.permit() performs the signature validation using `ecrecover()`, it is recommended to require the given signature to be canonicalized (e.g., as in the ECDSA library) as a preventative measure for signature malleability.

```
require(owner == ecrecover(digest, v, r, s), "ERC20: invalid-permit");
```

ERC20Permit.sol#L210

Status

Fixed in PR 184.

B10: Fake pool creation

[Severity: Informative | Difficulty: N/A | Category: Security]

ConvergentPoolFactory.create() can be called by anyone. A malicious user can exploit this to create a fake pool to deceive benign users.

Scenario

1. Suppose the Element Finance team is going to create a pool.
2. Knowing the plan in advance, Alice creates a fake pool with the same underlying token, name, and symbol, but with a fake bond token that looks very similar to the legitimate one. The ConvergentPoolFactory contract will emit both PoolRegistered and PoolCreated events with the fake pool and bond.
3. Then, Alice deceives other users to sell their underlying token and buy the fake bond token.
4. Furthermore, some arbitrageurs (bots) who are monitoring the PoolRegistered and PoolCreated event emission may rush and buy fake bond tokens at the initial price that they believe is a discount.

Recommendation

Add authorization logic to ConvergentPoolFactory.create() as well as other factory functions, e.g., TrancheFactory.deployTranche().

Status

Acknowledged by the client.

B11: Documentation for Tranche.prefundedDeposit()

[Severity: Informative | Difficulty: N/A | Category: Usability]

Tranche.prefundedDeposit() requires the funds to be transferred to the position contract in advance. However, it is critical to have the fund transfer and the prefundedDeposit() call to be executed within a single transaction, otherwise the transferred fund could be lost (i.e., deposited in another user's name). For those who may want to have a custom integration with Tranche instead of going through UserProxy, it is recommended to well document this behavior (e.g., in the natspec comment) to prevent any potential misuse of this function and loss of their funds.

Status

Fixed in PR 184.

B12: Risky trick

[Severity: Informative | Difficulty: N/A | Category: Security]

ERC20Permit.sol implements the ERC20 functionality for all principal tokens, interest tokens, and (wrapped) position tokens. Its constructor() function contains the following trick to automatically revert any transfer to either the zero address or the token address itself. This way, the transfer() and transferFrom() functions can save some gas by omitting the sanity check for the receiver address. However, it is concerned that this trick may introduce another attack surface that can utilize these "nonexisting" tokens. It is hard to completely rule out the possibility of potential hidden bugs (possibly of the external vaults integrated) being exploited to drain these nonexisting tokens. As such, it is recommended to have the explicit sanity check instead of taking the risk of this trick, considering that the gas cost for the sanctify check is small.

```
// By setting these addresses to 0 attempting to execute a transfer to
// either of them will revert. This is a gas efficient way to prevent
// a common user mistake where they transfer to the token address.
// These values are not considered 'real' tokens and so are not included
// in 'total supply' which only contains minted tokens.
balanceOf[address(0)] = type(uint256).max;
balanceOf[address(this)] = type(uint256).max;
```

ERC20Permit.sol#L44-L50

Status

Acknowledged by the client.

B13: Unused imports

[Severity: Informative | Difficulty: N/A | Category: Code Refactoring]

The following import is not needed and can be removed for readability.

```
import "../Tranche.sol";
```

InterestTokenFactory.sol#L4

Status

Fixed in PR 184.

B14: Open pre-approval of allowances

[Severity: Informative | Difficulty: N/A | Category: Security]

The UserProxy.mint() allows users to execute the permit() function of a number of arbitrary contracts. Although any concrete exploits have not yet been found, this openness is exposed to the potential for abuse in general. For example, a malicious token's permit() may utilize the allowance approved previously.

Status

Acknowledged by the client.

B15: Integration issues for WrappedPosition.prefundedDeposit()

[Severity: Informative | Difficulty: N/A | Category: Functional Correctness]

In the following code, `msg.sender` should be replaced with `_destination` so that the prefundedDeposit() function can be used in a different context later.

```
        uint256 balanceBefore = balanceOf[msg.sender];
```

WrappedPosition.sol#L111

Status

Fixed in 069ad749bc953366928deeac67f8da9c21e140a5.

B16: Potential division-by-zero errors in ConvergentCurvePool

[Severity: Informative | Difficulty: N/A | Category: Functional Correctness]

The pool initialization process consists of an initial liquidity deposit followed by a trade to set the initial price. If any liquidity deposits made before the initial trade will fail due to the division-by-zero error.

Scenario

1. Consider a pool for (Dai, fyDai). (Let us use the terminology of the YieldSpace paper.)
2. In the very beginning, Alice initializes the pool with 100 Dai, and receives 100 liquidity tokens. Now, the "virtual" reserve of this pool is (100 Dai, 100 fyDai), while the "actual" reserve is (100 Dai, 0 fyDai).
3. Bob joins the pool by adding 100 Dai. (Note that we suppose there is no trade made between (1) and (2).)
4. Then, the expected behavior is that: he should receive 100 liquidity tokens, and the new virtual reserve of this pool should become (200 Dai, 200 fyDai), while the actual reserve be (200 Dai, 0 fyDai).
5. However, the `_mintLP()` will fail to process Bob's joinPool() request due to the division by zero, since the actual reserve of fyDai (i.e., `reserveBond`) is zero at line 507.

Status

Acknowledged by the client.

B17: Sanity checks for ConvergentCurvePool

[Severity: Informative | Difficulty: N/A | Category: Fault Tolerance]

It is recommended to add a sanity check of "`require(_expiration - block.timestamp < _unitSeconds)`" in the constructor of ConvergentCurvePool, to early detect any potential mistakes with setting the parameters.

```
        expiration = _expiration;
        unitSeconds = _unitSeconds;
```

ConvergentCurvePool.sol#L94-L95

It is also recommended to add another sanity check of "`result > 0`" for `_getYieldExponent()` for extra protection:

```
        uint256 result = uint256(FixedPoint.ONE).sub(timeTillExpiry);
```

ConvergentCurvePool.sol#L633

Status

Fixed in d4223f4acfdcfd38005d53d925ddc8ad764dceca.

B18: Sanity checks for UserProxy.mint()

[Severity: Informative | Difficulty: N/A | Category: Fault Tolerance]

In UserProxy.mint(), it is recommended to have a sanity check that makes sure `_underlying` is associated with `_position` to early detect users' mistakes that can lead to losing their funds.

Status

Fixed in 0468d9e6f389daadf3be399afc2d762210b3cc3f.

B19: Sanity checks for YVaultAssetProxy constructor

YVaultAssetProxy would not work correctly when the number of underlying decimals is different from the number of vault decimals. Specifically, the following code would be broken if the two decimals values are different, because `_amount` has the underlying decimals, and `_pricePerShare()` is expected to have the underlying decimals:

```
        return (_amount * _pricePerShare()) / (10**vaultDecimals);
```

YVaultAssetProxy.sol#L202

(Note that if `_pricePerShare()` happens to have the vault decimals, then this function will work, but other functions will not.)

Moreover, the following code would be broken as well if the two decimals values are different, because the first return value of `_deposit()` has the vault decimals, but it is used to mint tokens of WrappedPosition.decimals() that is expected to be the underlying decimals.

```
        // Calls our internal deposit function
        (uint256 shares, ) = _deposit();
        // Mint them internal ERC20 tokens corresponding to the deposit
        _mint(_destination, shares);
```

WrappedPosition.sol#L87-L90

Although the current yearn vault decimals value is equal to the underlying decimals, that could be changed later. Also, in case that the two decimals values are different, it is not clear which decimals value vault.pricePerShare() would have. As such, for conservative protection, it is recommended to require the two decimals values are equal in the constructor of YVaultAssetProxy. If YVaultAssetProxy needs to be general enough to support different decimals values, additional analyses will be required.

Status

Fixed in 6c5c0e8fa57a539fc29d57d0a91557265485a219.

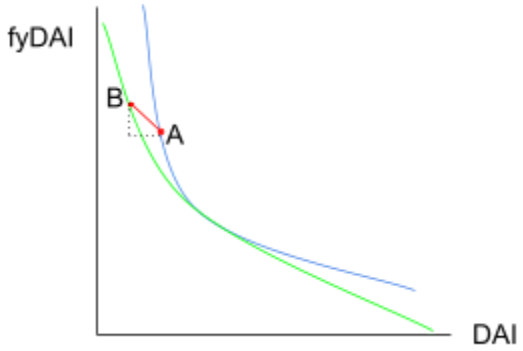
B20: Buying underlying tokens at discount with no fee

[Severity: Low | Difficulty: Low | Category: Security]

ConvergentCurvePool implements the YieldSpace curve. However, in the current implementation, there always exists an opportunity to buy underlying tokens at discount with paying *no* fee (other than the gas cost).

Scenario

In the following figure, suppose the blue curve denotes the pricing formula at $t = k$, and the green curve denotes the pricing formula at $t = k'$, where $k' < k$. That is, the pricing formula evolves from the blue curve to the green curve over time.



1. Suppose that at $t = k$ (i.e., the blue curve), the current reserve of DAI and fyDAI is at A.
2. Suppose that no trade has been made until $t = k'$ (i.e., the green curve).
3. Then, anyone can buy x DAI by selling only x fyDAI *without* paying any fee, such that $B = A + (-x, x)$. Note that there always exists such x in any given k and k'.
4. (Note that the current implementation charges no fee for this trade, because `amountIn` is equal to `amountOut` in `_assignTradeFee()` for this case.)

A more concrete scenario is as follows, visualized here.

1. Suppose `unitSeconds` is 1 year. Consider a 6 months lockup. That is, initially, $a = (1 - t) = 0.5$. (That is, $k = 0.5$.)
2. Suppose the initial reserve pair is (100 DAI, 100 fyDAI).
3. Suppose an initial trade is made to set an initial desired price, which updates the reserve to be A = (90.7 DAI, 109.754 fyDAI).
4. Suppose 1.2 months pass with no trades in the meantime. Now, $a = (1 - t) = 0.6$. (That is, $k' = 0.4$.)
5. Then, Alice can sell 1.124 fyDAI and buy 1.124 DAI. Now, the new reserve becomes B = (89.576 Dai, 110.878 fyDai).

Discussion

Note that this is different from typical arbitrage. An arbitrage trade is usually made to stabilize an unstable (spot) price. However, the above scenario happens even when A is the stable price at $t = k$. Note that the resulting price B is mostly *unstable* at $t = k'$, and thus there will exist further arbitrage opportunities to stabilize B. (The stable price at $t = k'$ is located at slightly below B along the green curve.)

In high-frequency trading pools, however, this exploit would not be beneficial because x would not be large enough to compensate for the gas cost. Thus, the severity would remain low in most cases.

Recommendation

Document this behavior due to the YieldSpace curve characteristics, especially for low-frequency trading pools, if any.

Status

Acknowledged by the client.

Test Coverage Analysis

The test coverage analysis powered by Firefly revealed missing test scenarios as follows.

No existing tests cover the case of "`percentFeeGov == 0`" in the following:

```
        if (percentFeeGov == 0) {
              // We reset this state because it is expected that this
              // resets the amount to match what's consumed and in the zero
              // that's everything.
              (feesUnderlying, feesBond) = (0, 0);
              // Emit a fee tracking event
              emit FeeCollection(localFeeUnderlying, localFeeBond, 0, 0);
              // Return the used fees
              return (localFeeUnderlying, localFeeBond);
        }
```

ConvergentCurvePool.sol#L621-L630

No existing tests cover the case of "`block.timestamp >= expiration`" in the following:

```
        uint256 timeTillExpiry = block.timestamp < expiration
              ? expiration - block.timestamp
              : 0;
```

ConvergentCurvePool.sol#L675

Recommendation

Add more tests to cover missing cases.

Status

Acknowledged by the client.