xTokens-uniswap

Initial:
https://github.com/xtokenmarket/xu3lp/commit/65ecb2806c4561869d0a744969428d13fd9a3ad8 (https://github.com/xtokenmarket/xu3lp/commit/65ecb2806c4561869d0a744969428d13fd9a3ad8)
Reaudit:
https://github.com/xtokenmarket/xu3lp/commit/4454137b14e1c9353bd87803b242497a6b715df6 (https://github.com/xtokenmarket/xu3lp/commit/4454137b14e1c9353bd87803b242497a6b715df6)

Contract does not work with tokens of different decimals

Severity: high
Status: Fixed

File(s) affected: All files

Description:
We found several issues when dealing with tokens of different decimals. When we talk about tokens being pegged, we talk about them in their decimal representation, i.e., `1.0 DAI == 1.0 USDC`, but as these tokens have a different number of decimals, this does not directly translate to the **token amounts** (in `uint256`). Meaning, `1 DAI != 1 USDC`, in fact `10^18 DAI == 10^6 USDC` as DAI has 18 decimals and USDC/USDT has 6 decimals. The contract however treats each `token0` and `token1` amount the same which leads to critical issues throughout the code and requires a major refactor. The tests should be adapted as well to deal with the actual tokens and their decimals.

Some examples:

- `getTargetBufferTokenBalance`, `_provideOrRemoveLiquidity`, `calculateSwapAmount` can't just add these amount0 and amount1 because of different decimals
- `mintWithToken` fee bug: fee is in the other token but gets subtracted from amount. Recommendation: subtract fee from amount first
- `mintWithToken` one time `_mintInternal` is called with an amount in token0, othertimes in token1. This is fatal if a) decimals are different b) the tokens are not pegged. Recommendation: always convert to token0 amount and mint by token0 amount as `getNav` is also in token0.
- `getAsset0Price` default return value should take into account difference in decimals. `return 10^(token1.decimals - token0.decimals)`
- `burn`: `fee` is always in asset0 decimals, can't just increment token1 fees with this amount.

- `burn` : in the `transferOnBurn(outputAsset, transferAmount)` call, `transferAmount` is in asset0 decimals as it's derived from `totalBalance` . But it gets used as both asset0 and asset1, depending on `outputAsset` , inside the function.
- more issues throughout the code

Reaudit

The refactored code still has an issue with tokens of different decimals:

Reaudit: `getAsset1Price` is wrong for `decimals(token0) > decimals(token1)`

Severity: High
Status: Fixed in `3fbf670720c6ec3eeed0c6e4230a2184ff48d95d`

`tokenDiffDecimalMultiplier` can be the decimals multiplier from token0 -> token1 or token1 -> token0 depending on the which one of the tokens has more decimals. This makes this variable hard to use as it always requires rechecking the decimals of both underlying tokens to determine the direction of the scaling before using it.
This also leads to a bug where `getAsset1Price` does not work when `decimals(token0) > decimals(token1)` .

Imagine token0 having 18 decimals, token1 having 6 decimals:
The twap price is divided by `tokenDiffDecimalMultiplier = 10^12` : `(token1 / token0) / tokenDiff = 10^-24` .

Wrong TWAP

Severity: High
Status: Fixed

`Utils.getTWAP` computes `(10000/10001)^[priceDiff / secondsAgo]` but should compute `(10001/10000)^[priceDiff / secondsAgo]` according to Uniswap V3 white paper equation 5.5

`getAsset0Price` does not always return TWAP

Severity: High
Status: Acknowledged

Reaudit: The modified code always returns `1` and skips the TWAP if the latest oracle update happened in the block `observationTime >= currTimestamp` . Also the modified check `!Utils.lte(currTimestamp, observationTime, currTimestamp - lastObservationSecondsAgo)` seems to always evaluate to `false` as `!Utils.lte(...) =`

`observationTime > currTimestamp - lastObservationSecondsAgo = observationTime > currTimestamp - (currTimestamp - observationTime) = observationTime > observationTime = false`.
Furthermore, there is no constant time window on the TWAP anymore which can mean that TWAP prices might be done over a tiny window.

`getAsset0Price` does not always return the TWAP price when it should. It returns `1.0` if `observationTime > currTimestamp - TWAP_SECONDS` but `observationTime` is the *latest* (not oldest) oracle update. So if the oracle was updated within the TWAP window, it short-circuits to return `1.0`.
Usage of `Utils.lte` also always ends up in the first case of `(a <= time && b <= time)` because both values are less/equal than `currTimestamp`.

Can just always use `pool.observe`? Is it safe to assume that pool has been initialized for some time and oracle has been triggered before xu3LP provisioning starts on it?

Parsing of uint to int is dangerous

Severity: Low
Status: Fixed

- `Utils.subAbs` the parsing of uint256 to int256 is dangerous if it exceeds the max signed integer type, can just do `return amount0 >= amount1 ? amount0 - amount1 : amount1 - amount0`?
- `Utils.sub0` the parsing of uint256 to int256 is dangerous, can just do `return amount0 >= amount1 ? amount0 - amount1 : 0`?

`initialize` can be frontrun

Severity: Low
Status: Acknowledged

Needs to be called right after deployment and verified that the transaction succeeded.

burn seems to be broken

Severity: Undetermined
Status: Acknowledged: the multiplication by the price when minting / burning is intended to prevent arbitrages

Reaudit: Consider this scenario where one of the prices (either asset0Price or asset1Price by symmetry) is higher than 1.0 like `1.05`.
User mints initial xU3LP tokens.
When user would redeem the `totalSupply` of xU3LP tokens, the `proRataBalance =`

`totalBalance * totalSupply * 105/100 / totalSupply = totalBalance * 105/100` will be higher than the `totalBalance`, therefore user only needs to redeem less than 100% to reach the full `totalBalance`. This shouldn't be the case as with mint followed by a burn should always return less or equal tokens.

Clarify for `burn` what token the `amount` parameter is in. It would make sense for it to be in the xu3LP token (as standard burn functions are) which the `super._burn(_, amount)` call also suggests. However, the `getAmountInAsset0Terms(amount)` call assumes that `amount` is in token1. Shouldn't it just be `proRataBalance = totalBalance.mul(amount).div(totalSupply())`, i.e., `proRataBalance = lpBurnAmount / lpTotalSupply * totalBalanceInAsset0`?

Wrong fee increase

Severity: high
Status: Fixed

`getBufferToken1Balance` subtracts `withdrawableToken0Fees` instead of `withdrawableToken1Fees`

Clarify how rebalance works

Severity: Undetermined
Status: Acknowledged, Info

The `_rebalance` and `checkIfAmountsMatchAndSwap` seem very complicated with several rebalance loops and are supposed to work in all four scenarios of having to mint/burn token0/token1, see the use of `subAbs` in `_provideOrRemoveLiquidity`.
Clarify how they work and the formula used in `Utils.calculateSwapAmount`.

Claim fees not always applied

Severity: Medium
Status: Mitigated: `withdrawAll` collects claim fees, `_unstake` does not.

`_unstake` / `withdrawAll` collect fees but do not take the `claimFee` cut. `withdrawAll` fee already collects all fees and the second `_collect` call will not collect any fees anymore?

Use SafeERC20

Severity: High
Status: Fixed

Should use `SafeERC20` everywhere for USDT support, see `mintInitial` where it does a normal `.transferFrom` call.

Reaudit: Contract only works for tokens with at most 18 decimals

Severity: High
Status: Acknowledged - the initial version will only use tokens with no more than 18 decimals

Reaudit: `getAsset1Price` returns asset 0 price

Severity: High
Status: False positive

The `getAsset1Price` returns `token1 / token0` (in a scaled way), i.e., the asset0 price which is used to convert from asset0 to asset1.

> `pool.observe` : "The time weighted average tick represents the geometric time weighted average price of the pool, in log base sqrt(1.0001) of token1 / token0." - Uniswap V3 inline docs

In the contract it's now used in the opposing way: `getAmountInAsset0Terms` uses `getAsset1Price` to convert asset1 to asset0. ("Returns amount in terms of asset1"). Its important to note that this does also not scale magnitudes in any way and the output will have the same number of decimals as the input.
Vice versa with `getAmountInAsset1Terms` .

Miscellaneous

- Fixed: `getStakedBalance` : first call could be to `getStakedTokenBalance` instead of implementing its function body again
- unlocked pragma
- Acknowledged: `initialize` sorts token0/token1 manually. while this is correct, these are internal implementation details of Uniswap. should just call pair.token0() and pair.token1(). And derive the `pair` from token0 and token1 first and not as an argument to make sure the pair matches the tokens
- Acknowledged: `mintWithToken` should take the token as an address (and check that it's either token0 or token1) to avoid depositing the wrong one. always makes integration easier as the caller site does not have to sort the tokens
- Fixed: `SWAP_TIMEOUT` seems unnecessary for the `swapTokenXForTokenY` functions. If block timestamp is used it always succeeds as router checks `deadline >=`

`block.timestamp` itself.

- Fixed (Reaudit): `MINT_BURN_TIMEOUT` is not needed
- Reaudit: It's now unclear which functions take amount parameters in *wei* precision and which ones work on the underlying token precision as all variables are just called `amount`. For example, `checkIfAmountsMatchAndSwap` and `_stake` work on the underlying precision while most of the other functions work with the *wei*-scaled amounts. This can easily lead to mistakes when calling the function. Choose a more descriptive naming convention that indicate the precision of the variables.