OpenZeppelin | security

Synthetix V3 Phase 2 Audit

SYNTHETIX

April 26, 2023

This security assessment was prepared by OpenZeppelin.

Table of Contents

Summary

Type	DeFi
Timeline	From 2022-02-22 To 2022-03-28
Languages	Solidity

Total Issues	70 (34 resolved, 15 partially resolved)
Critical Severity Issues	7 (2 resolved)
High Severity Issues	8 (2 resolved)
Medium Severity Issues	4 (2 resolved, 2 partially resolved)
Low Severity Issues	21 (11 resolved, 4 partially resolved)
Notes & Additional Information	30 (17 resolved, 9 partially resolved)

Scope

We audited the `Synthetixio/synthetix-v3` repository at the `f4081865557a0d650c3bac7e05641d797ff345e1` commit.

In scope were the following contracts:

```
./markets/spot-market/contracts/
├── Proxy.sol
├── interfaces
│   ├── IAsyncOrderConfigurationModule.sol
│   ├── IAsyncOrderModule.sol
│   ├── IAtomicOrderModule.sol
│   ├── IFeeConfigurationModule.sol
│   ├── ISpotMarketFactoryModule.sol
│   ├── ISynthTokenModule.sol
│   ├── IWrapperModule.sol
│   └── external
│       ├── IChainlinkVerifier.sol
│       ├── IFeeCollector.sol
│       └── IPythVerifier.sol
├── modules
│   ├── AsyncOrderConfigurationModule.sol
│   ├── AsyncOrderModule.sol
│   ├── AtomicOrderModule.sol
│   ├── CoreModule.sol
│   ├── FeatureFlagModule.sol
│   ├── FeeConfigurationModule.sol
│   ├── SpotMarketFactoryModule.sol
│   ├── WrapperModule.sol
│   └── token
│       └── SynthTokenModule.sol
├── storage
│   ├── AsyncOrder.sol
│   ├── AsyncOrderClaim.sol
│   ├── AsyncOrderConfiguration.sol
│   ├── FeeConfiguration.sol
│   ├── Price.sol
│   ├── SettlementStrategy.sol
│   ├── SpotMarketFactory.sol
│   └── Wrapper.sol
└── utils
    ├── FeeUtil.sol
    ├── SynthUtil.sol
    └── TransactionUtil.sol
```

```
./protocol/oracle-manager/contracts/
├── Proxy.sol
```

```
├── interfaces
│   ├── INodeModule.sol
│   └── external
│       ├── IAggregatorV3Interface.sol
│       ├── IExternalNode.sol
│       ├── IPyth.sol
│       └── IUniswapV3Pool.sol
├── modules
│   ├── CoreModule.sol
│   └── NodeModule.sol
├── nodes
│   ├── ChainlinkNode.sol
│   ├── ExternalNode.sol
│   ├── PriceDeviationCircuitBreakerNode.sol
│   ├── PythNode.sol
│   ├── ReducerNode.sol
│   ├── StalenessCircuitBreakerNode.sol
│   └── UniswapNode.sol
└── storage
    ├── NodeDefinition.sol
    └── NodeOutput.sol
```

```
utils/core-modules/contracts/
├── Proxy.sol
├── interfaces
│   ├── IAssociatedSystemsModule.sol
│   ├── IDecayTokenModule.sol
│   ├── IFeatureFlagModule.sol
│   ├── INftModule.sol
│   ├── IOwnerModule.sol
│   └── ITokenModule.sol
├── modules
│   ├── AssociatedSystemsModule.sol
│   ├── CoreModule.sol
│   ├── DecayTokenModule.sol
│   ├── FeatureFlagModule.sol
│   ├── NftModule.sol
│   ├── OwnerModule.sol
│   ├── TokenModule.sol
│   └── UpgradeModule.sol
└── storage
    ├── AssociatedSystem.sol
    ├── DecayToken.sol
    ├── FeatureFlag.sol
    └── Initialized.sol
```

System Overview

The audited codebase is composed of three main packages: the spot market, the oracle manager, and a set of utility core modules that are used throughout the codebase.

Spot Market

This system generates synths (which are ERC-20 tokens) and their respective markets (where they trade using the snxUSD stablecoin). Prices are determined by the oracle manager after applying a variety of fees. Synths track the prices of their underlying assets through oracle feeds. Users can trade without extra slippage due to price impact, as is the case with current AMM models.

To execute orders, markets withdraw and deposit snxUSD within the Synthetix core protocol, negatively or positively impacting the debt positions of liquidity providers, respectively. The total synth supply (subtracting the value of any collateral deposited in the markets) is reported as debt, fully collateralized by Synthetix V3 liquidity providers, who cannot undelegate their collateral if this were to leave the synth markets undercollateralized.

Transaction Types

The spot market system supports three types of transactions:

- Atomic orders: these enable traders to buy or sell synths in a single transaction, at the price reported by the oracle at the time of execution. This is equivalent to performing a swap on an AMM DEX, but the execution price is guaranteed to be the one reported by the oracle, regardless of the order's size.
- Asynchronous orders: designed to offer lower fees, these transactions use a two-step process (commitment and settlement) to prevent front-running. When traders commit to buying or selling a specific synth, the funds are transferred into escrow until the settlement window opens. A third-party permissionless actor, known as the keeper, will be able to settle the committed orders in exchange for a settlement reward, completing the trade for the initial user. If the order cannot be settled at a price that satisfies the trader's slippage settings, it can then be cancelled upon expiration and a full refund will be issued.
- Wrapping and unwrapping: if a market allows collateral wrapping, traders can deposit their native tokens and receive the equivalent synth in exchange, effectively lowering the

market's risk since the newly-issued synths are fully collateralized by their underlying assets. Unwrapping effectively burns the synths in exchange for the underlying collateral.

Various fees and configurations are available to generate profits and enable risk management for liquidity providers. These include utilization rate fees, auto-rebalancing skew fees, custom fee collectors, and interest rates. Market owners can configure these settings to optimize their markets' performance.

These fees can be partially or fully collected by a custom fee collector, and be distributed arbitrarily. If fees are not set, or if there are uncollected fees, all proceeds are paid out to liquidity providers (by depositing them into the market), effectively lowering their positions' debt.

One novelty introduced to incentivize the support of assets with a lower trading frequency is the concept of decay. The decay will rebase the token balances down with an increasing variable velocity that is reset at each trade. Thanks to this, if there is no trading activity for a long time, liquidity providers will be compensated for the lack of fees by having their debt reduced over time. This is useful for assets that are considered to be stable stores of value, such as gold.

Oracle Manager

The oracle manager package is in charge of configuring both individual oracles and complex systems with multiple oracle feeds, but it is also responsible for processing price queries and providing the results to the caller. Oracle configurations can be simple (i.e., fetching Chainlink's oracle feed) or complex (i.e., an oracle that retrieves the average price between two simple oracles).

The main contract is the `NodeModule` (which routes calls through different feeds), composed of several types of nodes. Nodes are the basic building blocks of the oracle manager. Node configurations can be defined by anyone and have a validation function that enforces the correctness of the initial parameters. Price retrievals are performed through the `process` function. Prices are always reported with 18 decimals. Some different types of nodes include:

- The `UniswapNode`: a simple Uniswap v3 TWAP oracle.
- The `ChainlinkNode`: a simple Chainlink price feed.
- The `PythNode`: a simple Pyth price feed.
- The `ReducerNode`: a complex oracle composed of two or more oracles. It is able to return a price that can be the `min`, `max`, average, median, `mul` or `div` of the retrieved prices, or just simply the most recent price. Anyone can build complex derivatives thanks

to these combinations, such as a synth tracking the price of ETH squared (to open a leveraged ETH position).
- The `PriceDeviationCircuitBreakerNode`: a complex node that ensures the reported prices do not deviate by more than a defined threshold between each other, providing an optional fallback node for when this occurs.
- The `StalenessCircuitBreakerNode`: a complex node that ensures the reported prices have not been reported on a date that is earlier than a defined threshold (i.e., stale), providing an optional fallback node for when this occurs.
- The `ExternalNode`: a general-purpose node that interacts with an external contract. The only requirement for the external contract is that it adheres to a defined interface.

Core Modules

The core modules are a set of auxiliary modules used by the spot market and the protocol. See their descriptions below:

- Associated systems module: the core protocol is accessed through a main router, and all of its components can communicate with each other. However, interacting with external systems is sometimes necessary. This module registers these systems to manage these interactions through the same router. An example of an associated system is the snxUSD stablecoin.
- Decay token module: an ERC-20 token module with downward rebasing functionality. It is used to allow synth owners to set interest rates by reducing the total supply of synths over time. As such, it is clear that synth tokens extend from the decay token module.
- Feature flag module: allows the owner to enable and disable features from being used by particular addresses, or in general. Currently, the only feature-flagged feature in scope is the creation of synths.
- NFT module: an ERC-721 token module. It is used in the protocol's account module to represent accounts as transferrable NFTs.
- Owner module: this module provides ownership initialization and management features.
- Token module: provides a standard ERC-20 implementation and can be used as a synth implementation.
- Upgrade module: allows the owner to upgrade the UUPS implementation.

Privileged Roles

The following actors are considered to have privileged roles within the system since they can perform special actions that can influence the end-users. Here is a summarized list:

- The system `owner`, which is the Synthetix team's multi-sig wallet, can:
  - Upgrade the implementation.
  - Leverage the FeatureFlag module to configure which features are available to specific users, or even to everyone if they see fit.
  - Manage deniers, who are individual addresses that have the power to unilaterally disable features.
- Deniers can disallow the creation of synths for everyone.
- Synth creators can create synth markets and assign ownership to any address they see fit (not necessarily their own).
- Synth owners can perform the following actions on their synth market:
  - Change the price feeds for buying and selling.
  - Upgrade the synth implementation.
  - Add new settlement strategies and decide whether they are enabled or disabled at any given time.
  - Change the fee configurations.
  - Change the fee collector.
  - Set custom fixed fees for individual addresses. These fees can have arbitrary values, but not zero.

Trust Assumptions

Given the number of privileged roles within the system, there are many trust assumptions that should be highlighted and must hold true as long as the aforementioned privileged roles are maintained. There can be negative side effects if one of the following trust assumptions no longer holds:

- Every trader is subject to fees, but synth owners may override the fixed fee to any non-zero value. It is assumed that a synth's owner will not use their `feeCollector` to avoid paying fees, as this will disincentivize users from trading with their synth and hurt the liquidity providers in the core protocol. In general, the synths owners are expected to never act maliciously.
- It is assumed at all times that synths have 18 decimals, and the same must be true for the underlying assets that are set in the `WrapperModule`. This second assumption is related to C03 but even if that issue is resolved, the protocol will always assume that

system amounts have the same number of decimals as the synth tokens. If either of these assumptions is violated, price retrieval from oracles and wrapping/unwrapping features may start accounting with an erroneous number of decimals.

- Unmanaged associated systems are not necessarily behind a proxy, so they are expected to be trustworthy, and it is particularly important that their storage is not manipulated externally.
- Associated systems can be upgraded at any time to another version, changing the underlying logic.
- The owner of the associated systems is the protocol itself, but there is no specific usage of the `OwnerModule` within the associated systems. Because of this, their ownership cannot be transferred or renounced in any way.
- In the `NftModule`, the owner can `burn` and `setAllowance` for any `tokenId`.
- `feeCollector` is expected to pull snxUSD as fees when called. If instead of pulling funds, a malicious implementation was to send any amount of snxUSD back into the market contract, synth trading would revert due to an underflow, and thus brick that specific synth market. In general, the `feeCollector` is expected to do nothing but accept up to 100% of the fees, since they are also capable of re-entering the spot market (see H05).
- Oracle prices are cast to `uint`, so the trading of assets with negative prices is disallowed (e.g., oil prices at some point in time).
- When calculating fees, `getMarketCollateral` will use the price reported by the oracle that is configured in the core protocol, while `getMarketCollateralAmount` will retrieve a collateral amount that is then multiplied by the price of the oracle configured in the spot market. It is assumed that discrepancies between the two oracles are negligible.
- Any change to the time windows in the settlement strategy can potentially affect any unsettled off-chain Pyth orders while the price is being validated.

Update: *Our assessment of the fixes implemented in the codebase is non-exhaustive, as the Synthetix team has indicated that a significant number of the identified issues will be addressed during the forthcoming refactor.*

Critical Severity

C-01 Async sell trades are exposed to inflation attacks

Async trades can be of two types: `buy` or `sell`. When selling, synth amounts are used. Since the synth tokens extend from the `DecayTokenModule` and async trades require some time to be either settled or cancelled, the decay must be taken into account. For this reason, async sell trades calculate the amount of shares corresponding to the given amount of synths, to keep track of the shares in escrow.

In order to account for synths' token balances and calculate the number of shares corresponding to each synth amount, the `AsyncOrder` library calls `balanceOf(address(this))`, which unfortunately opens the door to inflation attacks. Consider the following scenario:

- The escrow service has no activity yet. An attacker decides to commit an async sell order. If a `settlementReward` has been set for the specified settlement strategy, then the attacker will commit the minimum amount necessary to bypass the validation checks (corresponding to a USD value of at least `settlementReward + 1` snxUSD).
- Since there has not been any activity yet, the same amount of shares will be allocated at a 1:1 ratio. The `totalEscrowedSynthShares` will be initialized to the same value. To render their order impossible to settle, the attacker can set `minimumSettlementAmount` to `type(uint256).max`.
- A legitimate trader now wants to commit an async sell order of an arbitrary amount of synths - say `50e18` synths.
- The attacker front-runs the trader's transaction by sending `50e18*(settlementReward + 1) - settlementReward` amount of synths directly into the escrow contract (`settlementReward + 1` is already held by the contract). Notice that this is enough for the `sharesAmount` calculation to become truncated to 0.
- The trader is now tricked, their funds are transferred into escrow and 0 shares are accounted for them within `totalEscrowedSynthShares`, but also during the `AsyncOrderClaim` creation.
- Now the trader can simply cancel their unsettleable order after it expires. In this timeframe, the attacker can repeat the front-run for every new async sell trade coming in

from other traders. If the attacker does not repeat this process though, the profits will just be shared amongst the traders who have not been exploited in the meantime.

- When the attacker cancels their order, the `transferFromEscrow` function is called with the initial committed shares amount. If the attacker repeats the front-run for any other user trying to commit an async sell order, they will take the entire balance kept in escrow at cancellation time. Otherwise, the amount received by the attacker will include the original committed amount plus the proportional amount of the profits that belong to them within the `totalEscrowedSynthShares`.

The final result however, independent of whether there were multiple attacks, is that the initial trader's escrowed amount is stolen, since 0 shares have been accounted for due to the inflation attack.

This is a known attack surface that involves complex potential solutions. Some workarounds include reverting when calculated shares are zero, using internal balance accounting instead of `balanceOf(address(this))` or minting *phantom shares* to make the attack more expensive. We suggest reading the following discussion to get more insights on the best potential solution for this particular project.

Update: *Acknowledged, will resolve in the spot-market's code refactor in pull request #1523. The Synthetix team stated:*

> *Note the added tests in the referenced pull request which simulate the inflation attack.*

C-02 Skew fee manipulation can lead to market draining

Among the set of fees implemented within the spot market, the skew fee is defined as the fee applied based on the ratio of outstanding synths to the skew scale. If no skew scale is set, then this fee is not applied.

The skew is calculated as the difference between the total synth value and the total wrapped collateral value. In turn, the total synth value is derived by adding up the total supply of synths multiplied by their current price and the `totalCommittedUsdAmount` value.

The final fee percentage is calculated as the average between `(skew +/- amount) / skewScale` (skew after accounting for current trade, `amount` will be added on buying orders and subtracted on selling orders) and `(skew / skewScale)` (initial skew before accounting for current trade).

Finally, the applied fee will be negative for selling orders and positive for buying.

If the fee is negative it will act in favor of the trader, who will receive a larger amount than expected out of the trade. An exploiter can create orders with slippage protection equal to `MAX_UINT`, so that this order will never get settled (it is impossible to satisfy these slippage settings). However, the committed amount will get escrowed and its value in snxUSD will be added to `totalCommittedUsdAmount`. This order, which is impossible to settle and can be cancelled upon expiration without incurring any risk, will increase `totalCommittedUsdAmount`, which artificially increases the total synth value formula defined above.

By increasing the total synth value, the skew can be manipulated to become disproportionately larger, proportionally to the amount of capital committed by the attacker. If the skew fee becomes artificially high, a very high negative skew fee will be applied to selling orders, which will drastically increase their output amounts.

The attack can be executed as follows:

- The attacker commits a very large async buy order to artificially inflate the total synth value, setting a slippage protection value that makes the order not possible to settle as described above.
- The attacker performs multiple atomic selling orders to extract as much value as possible from the delegated credit capacity due to a very high negative skew fee. This can be repeated until the entire credit capacity is drained from the market.
- Upon expiration, the attacker cancels their order to receive a refund of 100% of their capital.

Consider not including committed orders in the total synth value's calculation, since they are not guaranteed to settle and can expose the system to an attack such as the one described above.

Update: *Acknowledged, will resolve in the spot-market's code refactor in pull request #1491. The Synthetix team stated:*

> *`totalCommittedUsd` has been removed, which should resolve this issue.*

C-03 `WrapperModule` does not enforce consistency on token decimals

The `WrapperModule` contract is used to wrap and unwrap a specific collateral associated with a synth.

The synth owner is in charge of setting which collateral is related to the synth token and the maximum amount of collateral that can be wrapped. When wrapping and unwrapping, synth tokens are minted and burnt at a 1:1 ratio according to the underlying USD price.

When wrapping, the collateral is deposited into the core protocol. When unwrapping, it is withdrawn.

The internal accounting in the core protocol converts the amount of collateral to a system amount, which is an amount with 18 decimals, regardless of the native collateral decimals. However, the `WrapperModule` does not take this into account, which can be problematic.

Suppose that the collateral set for wrapping/unwrapping a specific synth has a number of decimals lower than 18 (e.g., BTC has 8 decimals):

- When wrapping, the current collateral deposited is retrieved from the core protocol, and denominated in the system's amount of decimals. If `maxWrappableAmount` is assumed to have `18` decimals, the check used by the system to verify that the wrapped amount does not exceed the limit will be skipped, since `wrapAmount` will have 8 decimals. If this happens, the `synthUsdExchangeRate` call will return a `wrapAmountInUsd` with 8 decimals, incorrectly assuming that they are 18 decimals. This will lead to an incorrectly low amount of synths being minted, potentially triggering a revert due to the check in line 94. If the assumption is that `maxWrappableAmount` has the same decimals as the collateral (in this case 8), then the "exceed" check will likely trigger itself unexpectedly, disabling the wrap feature.
- When unwrapping, the scenario is more troublesome. In this case, the correct amount of synths will be burnt, and the `returnCollateralAmount`, which is what is actually transferred to the user, will have 18 decimals instead of 8, effectively draining the entire collateral deposited into the market.

Consider refactoring the codebase to correctly scale the wrapped amounts and specify which decimal system the `Wrapper` struct is following. Additionally, consider refactoring the `Price` library to always account for collateral decimals, instead of assuming that they are always 18. Finally, consider appropriately scaling `returnCollateralAmount` when unwrapping.

Update: *Acknowledged, will resolve in the spot-market's code refactor in pull request #1514.*

C-04 Deposited collateral can be drained

Synth owners have the ability to arbitrarily set wrapping and unwrapping fees, which can be negative in order to incentivize users to wrap or unwrap collateral depending on how much exposure the market already has.

These fees have no restrictions on their value. They can be set arbitrarily, which can lead to an attacker potentially draining the entire collateral deposited in the market.

A wrapping operation, given an input amount of collateral `collateralIn`, will return an amount of synths calculated as `synthAmount = [collateralIn * price * (1 - wrapFee%)] / price`.

An unwrapping operation, given an input synth amount of `synthIn`, will return an amount of collateral out (`collateralOut`) calculated as `collateralOut = synthIn * price * (1 - unwrapFee %) / price`.

Joining both equations, an atomic profit can be made by wrapping collateral and immediately unwrapping it, provided that the following condition holds: `collateralOut / collateralIn > 1` (assuming transaction gas costs are negligible).

- `collateralOut = [collateralIn * price * (1 - wrapFee%) / price] * price * (1 - unwrapFee%) / price`
- `(collateralOut / collateralIn) = (1 - wrapFee%) * (1 - unwrapFee%)`

The price used is the `sellingFeedId` configured within the synth market, but it does not have an effect on the calculations since it is only used to calculate the snxUSD amount for the fees.

In order to prevent an attacker from draining the deposited collateral amount, when setting wrapper fees, consider enforcing that the equation `(1 - wrapFee%) * (1 - unwrapFee%) <= 1` holds true. Otherwise, an attacker can atomically loop a wrap and unwrap bundle of transactions in order to eventually drain all the deposited `collateralType` from the market.

Update: *Acknowledged, will resolve in the spot-market's code refactor at commit b5d7db4. The Synthetix team stated:*

> *We've added guardrails to the setter methods to prevent this scenario.*

C-05 Anyone can set the `DecayTokenModule` allowance

The `DecayTokenModule` is the base contract for the template used for synths. This is shown in the definition of the `SynthTokenModule` contract. The token itself is compliant with the ERC-20 interface. However, it features a `setAllowance` function that can arbitrarily set an allowance of any amount to any address, on behalf of any user.

The intention is for this function to only be called by the owner of the synth. However, the function is set as public, meaning anyone can call it.

Consider restricting access to the `setAllowance` function to prevent malicious users from approving unwanted allowances.

Update: *Resolved at commit e418ad3. The `setAllowance` function is now in the `TokenModule` and it has proper access control. The Synthetix team stated:*

> *`setAllowance` now inherits the correct implementation from `TokenModule`.*

C-06 Synths' delegated credit capacity can be drained

After a new synth market is created, the `synthOwner` can set different oracle feeds for buying and selling prices without restrictions. There is no enforcement that these prices will ensure the price reported for buying is higher than for selling (factoring in fees).

If two different feeds are set, both buying and selling prices will fluctuate and at some point, the selling price may become higher than the buying price. When this happens, an arbitrage opportunity may arise that will extract value from V3 stakers' delegating credit capacity on this specific synth market. The following execution flow, when executed atomically via a smart contract, can exploit this situation:

- An atomic order to buy a specific synth will return a `synthAmount` of synths, calculated as: `usdAmountIn * (1 - buyFees %) / buyPrice`.
- An atomic order to sell that same amount of synths will return a `usdAmountOut`, calculated as: `(synthAmount * sellPrice) * (1 - sellFees %)`.

Joining both equations with the common variable `synthAmount`, we get:

```
usdAmountOut = [[usdAmountIn * (1 - buyFees %) / buyPrice] *
sellPrice] * (1 - sellFees %)
```

The condition that can yield a profit for an exploiter is `(usdAmoutOut - transaction gas cost) / usdAmountIn > 1`. In order to simplify the calculations, we will assume gas costs are negligible, and simplify this to `usdAmountOut / usdAmountIn > 1`. This gives us the final condition to make a profit:

- `(sellPrice / buyPrice) * (1 - buyFees %) * (1 - sellFees %) > 1`, which is equal to:
- `(sellPrice / buyPrice) > 1 / [(1 - buyFees %) * (1 - sellFees %)]`

If all fees were zero, it is clear that an arbitrage opportunity can be exploited any time `sellPrice` is larger than `buyPrice`. However, even if the oracle-reported prices are guaranteed to be higher for the buying side and lower for the selling side (i.e., by using two reducer oracle nodes set to return the `max` and `min` values respectively for buying and selling), we still need to factor in the fees.

It is possible that the effective prices after fees are higher when selling than when buying, considering that selling fees may be negative or zero depending on market utilization and configuration, and buying fees can be zero. This means that the full equation described above should be checked if arbitrage wants to be avoided, rather than just the two price feeds.

If the necessary conditions described for this arbitrage present themselves at any point in time, the entire trade will be executed at a constant price (meaning there will be no price impact due to trade size as would happen on a decentralized exchange). This would allow an exploiter to profit from this opportunity atomically (in a single transaction), up to the total synth market's delegated credit capacity. This attack can be self-funded, performed with a flash loan or a combination of both executed in a loop.

The following example illustrates the conditions that would enable this attack:

- There is enough SNX as delegated collateral supporting the snxETH synth market from multiple stakers, to provide up to 1M snxUSD worth of credit capacity for the market to borrow against.
- Both buying and selling prices are equal since the synth owner has configured them to read from the same feed.
- The effective buying fee percentage is 0.1%.
- The effective selling fee percentage is -0.3%.
- The ETH reported price is 1000 snxUSD per ETH, for both buying and selling.

In this scenario, an attacker can drain the entire market capacity by leveraging a smart contract that performs the following actions atomically:

- Request a flash loan of snxUSD (or any other asset which is then swapped for snxUSD on the open market), resulting in a very large balance for the attacker (e.g., 100M snxUSD).
- Use the entire amount of snxUSD to atomically buy as many snxETH as possible. The output amount would be 100M * (1 - buyFees %) / buyPrice = 99,990 snxETH.
- Immediately use the same amount of snxETH to sell it for snxUSD. The output amount would be 99,990 snxETH * sellPrice * (1 - sellFees %) = 99,990 * 1000 * (1 - (-0.3%)) = 100,019,997 snxUSD.
- This cycle yielded a 19,997 snxUSD profit. The attacker can now execute this in a loop until the entire credit capacity is exhausted, making a profit on every iteration.
- Repay the flash loan.

The end result would be that the `creditCapacity` for that synth market would be fully utilized (in essence, fully cashed-out by the exploiter), leaving the V3 stakers with bad debt against their collateral.

As a final note, this attack can also be achieved by using the `WrapperModule` to substitute the buying leg of the trade. This means that the attacker can decide whether it is best to wrap ETH to get snxETH, or if instead they should purchase snxETH directly using snxUSD. They will choose the option that yields a better net effective price.

Consider enforcing a check before any atomic order that ensures this type of profit cannot be made in an atomic fashion at the expense of V3 stakers.

Update: *Acknowledged, will resolve in the spot-market's code refactor at commit 7f1ec77. The Synthetix team stated:*

> *We have added a check on atomic orders to ensure that the proceeds from a trade cannot be immediately exchanged again for a profit.*

C-07 `DecayTokenModule`'s `transferFrom` is flawed

The `DecayTokenModule` has a double representation of what an amount of tokens means. When minting new tokens, the `amount` being minted is converted to shares, according to the latest decay and volume. Shares are then minted and accounted for in the underlying ERC-20 contract.

However, when querying the `totalSupply`, a token amount is returned, reduced by the decay up to that point.

To account for the decay, the `balanceOf`, `transfer` and `transferFrom` functions are overridden. Specifically, the `balanceOf` function converts the shares held by an account and returns an `amount` of tokens by multiplying the shares by the amount of tokens per share (`_tokensPerShare`).

The `transfer` function, on the contrary, receives an `amount` of tokens, but the `shares` to which the `amount` is converted are transferred.

However, in the `transferFrom` function, the input is called `amount`, but is instead interpreted as `shares`. The reasoning is that the `amount` is multiplied by `_tokensPerShare`, which indeed represents an amount of tokens for an individual share. Subsequently, `transferFrom` assumes `shares` as input and transfers an `amount` of tokens, while `transfer` does the contrary, receiving an `amount` and converting it to `shares`, which are finally transferred.

Given the final accounting when minting is in `share` amounts, consider fixing the `transferFrom` function to behave like the `transfer` function and correctly account for the transfer, preventing unexpected amounts from being moved.

Update: *Resolved at commit c00e684 and pull request #1587. Two changes have been added along with the recommended fix; the `advanceEpoch` is now overwriting the `totalSupplyAtEpochStart` value, and the `_tokenPerShare` function is returning `DecimalMath.UNIT` if `totalSupply()` is zero.*

High Severity

H-01 Potential Overflow in `UniswapNode` Library

The team forked the `OracleLibrary` from Uniswap, which is used to retrieve prices from a Uniswap v3 pool. The first step in retrieving the price is to get the `tick` for the selected time window, and the final step is to call the `getQuoteAtTick` function to calculate the price of the asset. For example, to retrieve the price of ETH in USDC, the `quote` asset would be set to USDC.

The function works in the following order:

- Retrieve the square root of the price with the given tick.
- Check whether the square root of the price is smaller or greater than `type(uint128).max` using an `if/else` statement. This step is essential to prevent a possible overflow in the next step.
- If the square root of the price is smaller than `type(uint128).max`, perform an optimized calculation by multiplying the square root of the price by itself.
- If the square root of the price is greater than `type(uint128).max`, perform a slightly less efficient, but still optimized calculation that avoids the potential overflow.
- Calculate the final quote price based on the result of the previous step.

However, the audited codebase shows that the `if` statement was modified to use `type(uint256).max` instead. As a consequence, if the square root of the `quote` asset's price exceeds `type(uint128).max`, the calculation will overflow and the transaction will revert unexpectedly. Moreover, in this case, the `else` statement would never be called.

Consider changing the `type(uint256).max` condition to the original `type(uint128).max`. Alternatively, consider using the native `OracleLibrary` from Uniswap instead.

Update: *Resolved at commit d518885. As it stands, the `quoteAtTick` function is always called with a `baseAmount` scaled to 18 decimals, meaning it is assumed that the quote token always has this number of decimals.*

H-02 Keepers can sandwich settlements to steal trading fees

Keepers are permissionless actors with the ability to settle committed async orders in exchange for a reward (paid in snxUSD) provided that the settlement window is active (there is some delay enforced after commitment, before an order can be settled). Keepers have the privilege of being able to execute arbitrary code before and after the settlement takes place, effectively sandwiching async orders' settlement.

Async orders generate fees that will either be collected by the `FeeCollector`, or burnt in favor of the stakers by depositing them into the market, effectively lowering the debt for all pool participants.

A malicious keeper can deploy a smart contract to perform the following actions atomically when a committed async order is ready to be settled:

- Flash loan a very large amount of collateral (i.e., SNX).
- Deposit that amount into the core protocol.
- Delegate all collateral to a pool supporting the synth market being traded on the async order. By doing this, the new funds delegated will most likely be much larger than the current amount of delegated collateral thanks to the funds provided by the flash loan. This means that the attacker will own a very large percentage of the pool (i.e., 80%+).
- Settle the order. This will cause the fee to be burnt in favor of pool participants, lowering their debt. Since the attacker now owns the majority share of the pool, they will get a very large share of the fees for themselves, resulting in negative debt since the position was just opened.
- Mint as much snxUSD as possible so that the debt stays at zero (or alternatively, accumulate profits as negative debt).
- Undelegate all collateral, which succeeds since the debt is not positive.
- Withdraw all collateral from the protocol.
- Repay the flash loan.

If abused, the attacker can potentially drain the majority of the protocol's async trading revenue, thereby removing the incentives for stakers to participate in it.

Consider disallowing atomic delegations and undelegations by enforcing a minimum wait time of one block as a way to at least mitigate flash-loan-based attacks. Additionally, consider alternatives to avoid self-funded attacks that can potentially drain the protocol's revenue.

Update: *Acknowledged, will resolve in the spot-market's code refactor at commit a94bbed. The Synthetix team stated:*

> *SIP-320 mitigates this attack vector.*

H-03 Utilization rate fees can be manipulated

When calculating utilization rate fees inside a buy order, there are two variables (amongst others) that are taken into account to determine the outcome of the fee calculation. These are the `delegatedCollateral` and the `totalCommittedUsdAmount` variables. These two variables are external to the fee calculation. As such, a malicious actor can interact with the protocol to manipulate them and determine an artificial value for the utilization rate fees.

The `delegatedCollateral` variable can be forced to be high enough so that the `if` check in line 270 returns 0 for the utilization rate fee. This is possible since the collateral can be delegated and undelegated in the same block within the core protocol.

By performing the following actions, an attacker can skip the utilization rate fees (i.e., pay 0% utilization fee rate) without incurring any risk:

- Request a flash loan of the same collateral and delegate it into the core protocol. This will increase `delegatedCollateral`. Alternatively, one can create a large async sell order to deflate the `totalValueAfterFill` value and obtain the same result as hitting the `if` check.
- Place an atomic buy order. At this point, the utilization rate fees will be 0%.
- Undelegate the collateral and pay back the flash loan, or wait until expiration to cancel the async sell order and recover the escrowed funds.

The `totalCommittedUsdAmount` can also be artificially increased with an async buy order that will never settle (either by cancelling it or setting the slippage tolerance to `type(uint256).max` so that it will never be reached). If this occurs, utilization rate fees can be artificially increased so that either the core protocol stakers or the fee collector recipient receive higher fees in return, making a profit out of an innocent user that happens to perform an atomic buy trade after the `totalCommittedUsdAmount` has been inflated.

In order to avoid the first scenario, it may be necessary to disallow delegation and undelegation of collateral in the same block. This would not solve the problem entirely, but it would at least require assets to be locked while they are exposed to debt. In order to avoid the second scenario, consider not accounting for orders that are committed, and thus never settling in the `totalCommitedUsdAmount` variable. Finally, consider whether such fee manipulations can be avoided differently, ensuring that malicious actors cannot profit from them.

Update: *Acknowledged, will resolve in the spot-market's code refactor at commit 4f8b8cd. The Synthetix team stated:*

> `totalCommittedUsd` *has been removed, which should resolve this issue.*

H-04 `UniswapNode` returns 0 as timestamp

The `UniswapNode` is the oracle node in charge of retrieving the TWAP price from a Uniswap v3 pool. It works by first taking the `tick` and then calculating the quote corresponding to the retrieved `tick`, resulting in the asset price.

All oracle nodes must respect the `NodeOutput.Data` struct definition when returning a price through the `process` call. One of the parameters of the `NodeOutput` struct is the `timestamp`, which is supposedly the `timestamp` at which the price was last updated, even though this could have a different definition in other nodes (such as the `ReducerNode`).

However, the `UniswapNode` always returns 0 as a timestamp for the retrieved price, since Uniswap does not return this parameter, given that it is assumed that the TWAP price is retrieved from `secondsAgo` up to the current `block.timestamp`.

`UniswapNode`, along `ChainlinkNode`, `PythNode` and `ExternalNode` are basic oracle nodes. When combined, they can be used in more complex nodes such as the `ReducerNode`, the `StalenessCircuitBreakerNode` or the `PriceDeviationCircuitBreakerNode`. These last nodes have `parents` defined, which are likely instances of the former basic oracle nodes.

Returning 0 as a timestamp is not an issue on its own, but if the `UniswapNode` is used as a parent to other nodes, or if the code using it needs to operate on the `timestamp` returned, issues may arise.

- If set as the first parent of a `StalenessCircuitBreakerNode`, the node will permanently fall back to the second parent if available, and will otherwise permanently revert. This is because the first `if` in the `process` function of the `StalenessCircuitBreakerNode` will be never satisfied.
- If set as a parent in a `ReducerNode` configured with a `recent` operation, its price evaluation will be constantly skipped since its timestamp will always be less than or equal to any other parent's timestamp.
- If set as a parent in a `ReducerNode` configured with a `mean` operation, the final average timestamp will be artificially lowered, since it will count as an individual node in the total length when averaging, but will not contribute to the values of timestamps from

which the average is taken out. The same will happen if the operation configured in the `ReducerNode` is a `mul` or `div`.

Consider deciding whether the `UniswapNode` should return 0 as a `timestamp`, or rather `block.timestamp` or another meaningful value instead. Alternatively, consider refactoring the docstrings of composite oracle nodes such as the `ReducerNode` to warn users of the potential dangers of using a `UniswapNode` as a parent in their node configuration.

Update: *Resolved at commit 50957e5.*

H-05 Malicious synth creators can steal snxUSD from benign synths

For simplicity, suppose the spot market has a balance of 100 snxUSD due to a benign user committing an async buy order to buy 100 snxUSD worth of snxETH. A malicious synth creator (attacker) creates a synth called B with a 0 skew scale, 0% utilization fee rate, and a 100% fixed fee.

- The attacker atomically buys 100 snxUSD worth of B.
- Inside the atomic buy call, funds are pulled from the attacker, temporarily making the contract's balance 200 snxUSD.
- Fees are calculated to 100 snxUSD since there is a 100% fixed fee and all other fees are 0.
- The `collectFees` function makes a call to the fee collector. This is a malicious contract set by the attacker, since they created synth B.
- The malicious collector is approved to pull a maximum amount of 100 snxUSD from the spot market. After this pull, the market has a balance of 100 snxUSD (equal to the starting amount).
- After pulling the funds, and before ending the external call, the malicious contract re-enters the spot market and commits an async buy order of 100 snxUSD for snxETH (or any other benign synth).
- The contract's balance is now 200 snxUSD again, but the attacker has 100 snxUSD committed in their async buy order.

At this point, the following calculations take place:

- Collected fees are calculated as the difference between previous and current snxUSD balances, which resolves to 0.

- The amount of fees to deposit within the market is the difference between the total fees applicable for this trade and the amount collected by `feeCollector`. In this case, this resolves to 100 snxUSD, so that is the amount that gets deposited into the market.

In the end, the contract's balance is 100 snxUSD, the attacker has another 100 snxUSD escrowed in a buy async order, and yet another 100 snxUSD was deposited to the market manager for their malicious synth.

In this example, since the collector can re-enter and deposit the amount to an async buy order, this balance difference can be falsely turned to 0, pulling 100% of the fee twice (once to the attacker's async buy order and again to the market manager for the malicious synth), effectively stealing from the other synth's buy orders.

While users trust the owners of the synths they use, the setting of arbitrary contracts such as the `FeeCollector`, `oracle`, and `synthImpl` by malicious owners can pose a risk to benign synths, either because of reentrancy attacks or due to the modification of global state variables.

Consider adding the `nonReentrant` modifier from OpenZeppelin's reentrancy guard library to critical state-changing functions (especially functions related to atomic trading, wrapping, order commitment, and settling). In addition, consider using the checks-effects-interactions pattern wherever possible. Be mindful of other types of reentrancies, such as read-only reentrancy, and consider thinking about the potential implications of the `FeeCollector` interacting with external dependencies at this point in the execution flow.

Update: *Acknowledged, will resolve in the spot-market's code refactor in pull request #1524. The Synthetix team stated:*

> *We now push fees to the recipient rather than calling an external function to pull them.*

H-06 Async `SELL` orders cannot be cancelled

When an async order has passed its expiration time and has not been settled, it can be cancelled by the owner. In reality, anyone can cancel the order but they have no incentive to do so since there is no keeper fee implemented for order cancellations.

In the specific case of `SELL` orders, funds are returned to the trader via the `transferFromEscrow` function, which converts the amount escrowed from shares to the actual amount before executing the transfer of funds.

However, the actual transfer of funds is implemented using `transferFrom` instead of `transfer`. There is no self-approval before this point, so the transfer will revert due to having an insufficient allowance. Furthermore, there is no test to ensure funds are returned to the trader when cancelling an async `SELL` order (only when cancelling a `BUY` order).

Consider using `transfer` instead of `transferFrom` and adding a specific test for async sell order cancellations to ensure the funds are returned to the cancelling trader.

Update: *Acknowledged, will resolve in the spot-market's code refactor in pull request #1523.*

H-07 Skew fee calculations may unexpectedly fail

When traders wrap collateral to mint synths, they get slightly fewer synths than the collateral because of fees. This means that wrapping will add a little more value to the `wrappedMarketCollateral` than to the `totalSynthValue`. In addition, the total supply of synths can be constantly decreasing because of the decay token module.

For these reasons, a combination of traders minting synths by wrapping and time decay reducing their total supply can lead to a scenario where skew fees calculations may revert because of an underflow, having `wrappedMarketCollateral` greater than `totalSynthValue`.

This means that all atomic trades and async sell commitment transactions will revert until someone async buys or unwraps enough synths to make the `wrappedMarketCollateral` smaller than the `totalSynthValue` again. This situation can also be bypassed by setting the skew scale to 0, since the function will return 0 before the underflow happens, allowing for transactions to go through again.

Consider handling this case (where the `wrappedMarketCollateral` is strictly larger than the `totalSynthValue`) separately when calculating skew fees, in order to avoid unexpected reverts.

Update: *Acknowledged, will resolve in the spot-market's code refactor in pull request #1501. The Synthetix team stated:*

> *The skew fees now work as expected and deal with negative values appropriately. Relevant tests have been added to the codebase.*

H-08 Inconsistent fee calculations

The total escrowed value in USD is tracked using the global storage variable `totalCommmittedUsdAmount`. This variable accounts for all buying and selling committed orders by adding or subtracting their collective USD value respectively, as if those orders had already been settled (to account for potential deltas in supply due to async trading). This variable is also used as part of the utilization and skew fee calculations.

When traders commit async buy orders, their snxUSD amounts get locked in escrow and added directly to the `totalCommittedUsdAmount`, while all committed sell orders subtract the snxUSD value of their escrowed synth amounts. This value is checked via oracle at the commitment time.

The problem is that the USD value for sell orders at the time of commitment does not necessarily equal the USD value at the time of settlement. In other words, if there are price fluctuations between commitment and settlement, the actual settled USD amounts will differ from committed USD figures.

In fact, as prices fluctuate, multiple sell orders will contribute to the `totalCommitedUsdAmount` variable with different amounts, and will be mixed with the actual USD amounts from buy orders.

As a result, any atomic trading that takes place before the settlement window starts will be exposed to arbitrarily different utilization and skew fee calculations. Furthermore, if async orders never settle and get cancelled instead, atomic traders will be either undercharged or overcharged.

Consider storing committed amounts separately for buy (as USD amounts) and sell orders (as synth shares). To perform fee calculations consider setting `totalCommittedUsdAmount` to the committed USD from buy orders minus the committed synth shares from sell orders at the settlement time's price.

Additionally, consider removing the `totalCommittedUsdAmount` value from fee calculations altogether, since those orders may never settle and thus, users may end up being overcharged (see C02).

Update: *Acknowledged, will resolve in the spot-market's code refactor at commit 4f8b8cd. The Synthetix team stated:*

> *`totalCommittedUsd` has been removed, which should resolve this issue.*

Medium Severity

M-01 Synth tokens cannot have the `decayRate` set

Synth tokens are created through the `SpotMarketFactoryModule`. The `SpotMarketFactoryModule` has a feature flag to allow specific actors to create synths, and when one is created, the `synthOwner` input parameter is set as the account in charge of:

- Updating the price feeds for the synths
- Upgrading the logic implementation of the token to a new one
- Nominating a new owner

However, when the synth is created for the first time, the real `owner` of the synth token itself is the proxy behind the spot-market package, which also contains other modules, such as the `AssociatedSystemModule`.

The `AssociatedSystemModule` deploys a proxy behind the configured token implementation, and initializes it when it is first deployed. The synth token is an instance of a `DecayTokenModule` contract, meaning that owner-restricted features can be called exclusively by the deployer, which is the proxy of the spot market.

Altogether, this means that functions like `mint` and `burn` can only be called within the spot market, and not externally. This can be observed in `AsyncOrderModule`. The `mint` and `burn` calls are actually external calls toward the synth token proxy, whose owner is the spot market.

However, within the codebase, there is no function that calls the `setDecayRate` function of the `DecayTokenModule`, so there is no way to set it. The spot market implementation should be upgraded to include a function that calls `setDecayRate` internally in order to appropriately set the `decayRate` of the synth token. Alternatively, the `synthOwner` could upgrade the synth implementation to allow the `synthOwner` to set the `decayRate`, instead of only allowing the synth token owner to do so.

Consider allowing the `synthOwner` to call the `setDecayRate` function.

Update *Resolved at commit 62ebf1c. The team also made the `upgradeSynthImpl` function public, since the owner can now just change the synth implementation and anybody can upgrade specific synths to the latest implementation.*

M-02 Oracle data is not validated

Many oracle node implementations (in both `validate` and `process` functions) are not checking whether the price returned by the oracle is nonzero or that the answer, apart from not reverting, returned valid data. Moreover, the `core` protocol always converts prices to `uint` variables, while prices are allowed to be `int` instead. If negative prices are returned, the `uint` conversion will fail.

Finally, the `PriceDeviationCircuitBreakerNode`, the `StalenessCircuitBreakerNode`, and the `ReducerNode` are not checking whether their configured parents are processable at their `validate` function, meaning that a call to their `process` function does not revert.

As mentioned in M04, consider deciding whether zero-value prices should be allowed, and either reflect this in the docstrings or add a check in the codebase. Moreover, consider deciding whether negative prices should be accepted in some instances, and add a method to properly handle them at the protocol level or within third-party components. Additionally, decide whether the `validate` function should check for the ability to process parents' feeds. Finally, in some circumstances the `validate` function returns data that is not the price, such as the cumulative `tick` returned by the Uniswap node. Still, the only checks in these cases relate to the call not reverting, and the validity of the returned data is never verified.

Update: *Partially resolved at commit eca92a3. The `ReducerNode`, `PriceDeviationCircuitBreakerNode` and `StalenessCircuitBreakerNode` are now checked to have processable parents.*

M-03 Missing validation in Uniswap Oracle's configuration

The `UniswapNode` library defines the `validate` and `process` functions of an oracle using the Uniswap V3 TWAP oracle feature.

In order to save gas by not calculating redundant variables, the team has opted to only port specific functionality into a custom library instead of relying on the Uniswap-provided `OracleLibrary` contract. Specifically, when querying the TWAP price from a Uniswap V3

oracle, one initially calls the `consult` function, specifying in which `pool` and how many `secondsAgo` we would like to calculate the TWAP price. The `consult` call returns the arithmetic mean tick between `secondsAgo` and the execution time, as well as the harmonic mean liquidity. The harmonic mean serves no purpose in the library, leading the team to only port what was needed.

This part corresponds to lines 37 to 49 of `UniswapNode.sol`.

As we can see, the check for `secondsAgo` to be non-zero (since it would later lead to a division by zero) has been removed. However, in the Synthetix codebase, this parameter is part of the oracle configuration, permanently set in storage.

At first impression, one might think that this is checked once the oracle is set up, in the `validate` function. However, this is not the case. Uniswap oracles can be set with `secondsAgo == 0`, in which case the oracle would revert at any `process` call. Notice that setting `secondsAgo == 0` might be thought of as retrieving the latest price without a TWAP mean, but the current codebase does not allow for that.

Consider adding a check for `secondsAgo != 0` in the `validate` function of the `UniswapNode` contract. Alternatively, consider using the native `OracleLibrary` from Uniswap instead.

Update: *Resolved at commit c1fd8c6.*

M-04 Zero prices may be handled incorrectly

Complex oracle nodes use parents as their price source, and later apply some complex logic on top of the returned parent prices. However, having a price of zero does not necessarily mean that something is wrong with the oracle, given that prices are never checked to be non-zero in the nodes' `validate` functions. If an asset's price goes to zero, complex oracle nodes may be unable to properly handle this scenario.

In the `PriceDeviationCircuitBreakerNode`, if the first parent's price is zero and the third parent is not set, the oracle will always revert, while if a third parent is set, its price will be returned, regardless of its reported price value.

In the `StalenessCircuitBreakerNode`, if the second parent's price is zero and the principal parent's price is stale, the oracle will revert, while a non-stale zero price from the first parent will be returned as is, without reverting.

If one of the parents in a `ReducerNode` set to `mul` reports a zero price, the entire price returned by the `ReducerNode` will be zero.

In all three cases, it is unclear whether a price that is equal to zero should be allowed, as the oracle may behave differently depending on various conditions.

Consider analyzing each individual case, determining whether prices equal to zero should be allowed, and refactoring the code to be consistent with the intended behavior.

Update: *Partially resolved at commit 1247dca. The issue was addressed in the* `StalenessCircuitBreakerNode` *but not in the other nodes.*

Low Severity

L-01 Unsafe `uint` casting on Chainlink's off-chain settlement

When settling async orders using Chainlink as the off-chain strategy, the `offchainPrice` is calculated by casting the reported median price from the `IChainlinkVerifier` to `uint`.

Despite having a safety mechanism in place to enforce a maximum tolerance on price deviation between on-chain and off-chain prices, this casting is unsafe because it will not revert and may cause unexpected results.

Consider leveraging the `SafeCastI256` library to ensure no unexpected results occur.

Update: *Resolved at commit 4af6ab9. The team removed the off-chain settlement through Chainlink.*

L-02 Incorrect transactor value on async sell orders

Synth owners have the ability to configure custom fee collectors that implement the `IFeeCollector` interface. The goal is to allow them to collect variable fee amounts depending on different values, such as the trader's address or the order type.

The trader's address is passed onto the fee collector as the `transactor` address. On atomic operations, this value is set to `msg.sender`, which resolves to the address launching the trade. When settling an async buy order, `transactor` is set to the owner of the claim.

However, when settling async sell orders, this value gets assigned to `msg.sender`, which resolves to the keeper's address. The keeper is the operator that will settle the trade and earn a reward for it, but keepers do not necessarily need to be the original trader.

When calling `collectFees`, consider modifying this parameter to `trader`, which contains the async order claim owner. This will allow the fee collector to correctly determine the amount of fees that should be taken based on the trader's address.

Update: *Resolved at commit ba772ab.*

L-03 Duplicated code

There are instances of duplicated code within the codebase. This can lead to issues later on in the development life-cycle, and leaves the project more exposed to potential problems. Errors can be introduced when functionality changes are not replicated across all instances of code that should be identical. Some examples are:

- Within the `Price` library, there are two helper functions to retrieve the exchange rate for snxUSD against a given `synthMarketId`: `usdSynthExchangeRate` and `synthUsdExchangeRate`. In both instances, the current price needs to be retrieved, which can be achieved by calling `getCurrentPrice`. However, only the second function uses it, while the first one calls `getCurrentPriceData`, unnecessarily replicating the behavior of `getCurrentPrice`.
- The `abs` function is first defined in the `SettlementStrategy` contract and again in the `PriceDeviationCircuitBreakerNode` contract.
- `CoreModule` and `Proxy` are replicated in each package.

Instead of duplicating code, consider using the library or function containing the duplicated code whenever the duplicated functionality is required.

Update: *Partially resolved at commit 48c1b39. The team acknowledges that `CoreModule` and `Proxy` are still replicated.*

L-04 Incomplete spot market initialization is possible

The `SpotMarketFactoryModule` is considered initialized as long as the core protocol address and the snxUSD token address have been set.

However, both the oracle manager and the initial synth implementation might be left uninitialized (zero address), with no way to change them since these values cannot be updated after initialization.

Consider preventing the `SpotMarketFactoryModule` from being considered initialized until all four values are set to non-zero addresses.

Update: *Resolved at commit 67e6e18. The team removed the `InitializableMixin` dependency and any initialization feature in favor of specific setters for the implementation and the Synthetix address. We recommend adding specific unit test cases and ensuring that functions that assume proper initialization of initial values work as expected.*

L-05 Misleading synth implementation casting

Synth implementations are instances of `ISynthTokenModule`, which inherits from `IDecayTokenModule`.

However, when dealing with synths throughout the `SpotMarketFactoryModule`, it is always cast as `ITokenModule` which can be misleading since the actual implementations for all functions are different in both modules.

Consider refactoring the `AssociatedSystem` library so that it can also return instances of `ISynthTokenModule` when necessary.

Update: *Partially resolved at commit e418ad3. However, given the changes, there are now some inconsistencies like in the WrapperModule or the AsyncOrder, where `ITokenModule` is still used as the data type returned by the `SynthUtil.getToken` function. Consider reviewing the entire codebase to identify such occurrences and ensure that appropriate test coverage is in place to highlight potential issues caused by the use of the wrong interface.*

L-06 Incorrect index in `for` loop

The `for` loop in the `min` function of the `ReducerNode` library starts from 0 while it should be starting from 1, given that the first element is already processed outside the loop.

Consider starting the `for` loop from 1, similarly to the `max` function.

Update: *Resolved at commit af4dc62.*

L-07 Initializable modifiers are not used

Several contracts, such as `DecayTokenModule` and `TokenModule` extend from `InitializableMixin`. This contract offers the `onlyIfNotInitialized` modifier that can be used in the `initialize` function. However, this modifier is not being used in the contracts.

Moreover, both `DecayTokenModule` and `TokenModule` base their initialization status response on whether decimals are set. On the contrary, the `NftModule` contract uses the `Initialized` storage library to explicitly mark a contract as initialized, independently from token-specific variables.

Consider extending the use of the `Initialized` storage library to all initializable contracts, including `DecayTokenModule` and `TokenModule`. Moreover, consider making use of the `onlyIfNotInitialized` modifier in all `initialize` functions to prevent them from being called in future iterations.

Update: *Resolved. This is not an issue as the team decided to refactor the code differently as part of pull request #1586 at commit 6ae84fc. The `InitializableMixin` dependency has been removed from `DecayTokenModule` and the suggested modifiers are not being used. The team acknowledges these intentions and maintains the `initialize` function free to be called again in the future, under specific circumstances.*

L-08 Decay rate can be set to any value

The decay rate of the `DecayTokenModule` is supposed to have an upper bound of `1e18`. However, the `setDecayRate` function never checks for this bound when a new value is set.

If the decay rate is meant to go over 100%, consider explicitly explaining it in the docstrings. Otherwise, to avoid unexpected results when accounting for the decay, consider enforcing that the new value set for the decay rate stays within the limit of `1e18`. Be aware that if the `decayRate` is set too high, it may produce an unexpected revert in the `totalSupply` function, when the `_ratePerSecond` is being subtracted from `1e18`.

Update: *Resolved at commit 1bd0500.*

L-09 Missing initializer modifiers

Throughout the codebase, the `initialize` functions should only be callable once, during initialization. However, the following instances can be called multiple times:

- Line 49 of the `SpotMarketFactoryModule` contract
- Line 24 of the `DecayTokenModule` contract
- Line 31 of the `NftModule` contract
- Line 25 of the `TokenModule` contract

As already suggested in L07 and in the interest of predictability and clarity, consider applying the `initializer` modifier to guarantee that the `initialize` functions can only be called once, during initialization.

Update: *Acknowledged, not resolved. The team decided to refactor the code differently at commit 6ae84fc. The team acknowledges the intention of maintaining the `initialize` function free to be called again in the future, under specific circumstances.*

L-10 Lack of input validation

When creating a new synth, there are no checks in place to ensure that the `synthOwner` is not the zero address. If a synth was created with the owner set to the zero address, it would become unusable since it would be considered an invalid market.

Additionally, when updating price feeds for a certain synth, the new feed IDs are updated directly without validation. If incorrect IDs are used, all oracle updates will revert since the associated `NodeType` will be `NONE`, potentially disabling unhealthy positions from being liquidated.

Finally, when setting a settlement strategy or configuring fees, almost no parameters' values are verified.

Consider not allowing the `synthOwner` address to be the zero address when creating a new synth, ensuring new price feed IDs correspond to a valid oracle before updating their value, and verifying that all the relevant fields of any settlement strategy and fee configuration are set to appropriate values.

Update: *Partially resolved at commit 399a66f. Only the first item (`synthOwner`) has been resolved.*

L-11 Erroneous time window in Chainlink's oracle node

The `ChainlinkNode` oracle node processes a price request by first retrieving the `latestRoundData`, which also contains the `updatedAt` timestamp (when the price was last updated), and then calculating the `twap` through the `getTwapPrice` function.

The `getTwapPrice` function works by analyzing the round data between `startTime = block.timestamp - twapTimeInterval` and the current `block.timestamp`, and then calculating their average price.

However, it is not clear whether the timeframe to be analyzed should be `startTime = updatedAt - twapTimeInterval` instead, where `updatedAt` is the value retrieved by the

`latestRoundData` call in the `process` function. This means that if the oracle has not been updated for a time greater than or equal to `twapTimeInterval`, no rounds will be analyzed and the operation will be skipped, returning only the latest retrieved price.

Consider deciding whether this is the intended behavior, or changing the `startTime` calculation to cover the time window between `twapTimeInterval` and `updatedAt`, instead of `block.timestamp`.

Update: *Acknowledged, not resolved. The Synthetix team stated:*

> *We believe the expected behavior for the lookback window would be from the current time, rather than the time of the last update.*

L-12 Inconsistent oracle node validation

The `ExternalNode` oracle node, unlike Chainlink's, does not validate that a call to the oracle does not revert during its setup.

Moreover, the `ExternalNode` node does not enforce any `PRECISION`, as opposed to what is done by other nodes.

Additionally, nodes that are not external do not check that the target address enforces the associated interface (such as `IAggregatorV3Interface` for Chainlink or `IPyth` for Pyth). This is only the case for the `ExternalNode`.

Consider adding more validations to the `ExternalNode`, such as performing a `process` call inside the `validate` function or enforcing a certain `PRECISION` of 18 decimals. Additionally, consider ensuring that the relevant nodes enforce the associated interfaces. If these inconsistencies are intentional, consider adding docstrings to reflect this behavior.

Update: *Acknowledged, not resolved. The Synthetix team stated:*

> *We are unsure how we would validate precision, or what could be validated outside of the ERC-165 check.*

L-13 Spot market can be re-initialized

The `ISpotMarketFactoryModule` documentation states that the `initialize` function can only be called once, as long as it has not been initialized.

However, the actual `initialize` implementation does not check against a previous initialization, meaning it may be called multiple times with different values.

Consider enforcing that this function can only be called once, before the initialization.

Update: *Resolved at commit 67e6e18. The Synthetix team stated:*

> *We have removed the initialization pattern from the spot market.*

L-14 Uniswap oracle nodes are susceptible to manipulation on Optimism

As stated by Uniswap in their oracle documentation regarding integrations on Layer 2 roll-ups:

> On Optimism, every transaction is confirmed as an individual block. The `block.timestamp` of these blocks, however, reflect the `block.timestamp` of the last L1 block ingested by the Sequencer. For this reason, Uniswap pools on Optimism are not suitable for providing oracle prices, as this high-latency `block.timestamp` update process makes the oracle much less costly to manipulate.

For this reason, consider disallowing the creation of Uniswap nodes on Optimism by adding an extra check to the chain ID value upon validation, or explicitly stating the reasons for not doing so in the docstrings.

Update: *Resolved at commit eb9d8da.*

L-15 Inconsistent token decimals requirement in oracle nodes

Uniswap nodes will not pass the validation step if either of the tokens from the specified pool features more than 18 decimals.

However, in other nodes such as the `PythNode` or the `ExternalNode`, this is not enforced.

Additionally, when processing prices within a `UniswapNode`, the scenario where the price is reported with more than 18 decimals is actually downscaled, although it can never happen by design since the node would not have passed the validation step.

Consider allowing a `UniswapNode` to be valid if either of the tokens features more than 18 decimals to be consistent across the different node options. Alternatively, if this is intentional,

consider removing the downscaling logic when prices are reported with more than 18 decimals and explicitly describing the rationale in the docstrings.

Update: *Acknowledged, not resolved. The Synthetix team stated:*

> *The Uniswap Node is limited to using tokens with a maximum of 18 decimals places to avoid overflows in the `getQuoteAtTick` function. Downscaling is still necessary in the case of the stablecoin's decimals being greater than the token's decimals value.*

L-16 Incorrect function mutability

`pure` functions are not allowed to read contract state from storage - that is what `view` functions are for. However, due to a bug within the Solidity compiler, the case where a struct pointer is converted from a `storage` location into `memory` is not being caught correctly.

Due to the previous bug, some instances were found throughout the codebase where `pure` functions are being used to read state, such as the `getNode` function from the Oracle Manager package. This function returns a `memory` instance of `NodeDefinition.Data`, which is returned by the internal `_getNode` function. This internal function is also `pure`, returning a storage pointer to the location of the requested data.

In between these two functions, Solidity is reading the storage contents and loading them into memory, violating the nature of a `pure` function.

Since the pattern of passing storage pointers around in order to load data into memory is frequently used across the codebase, consider updating all `pure` functions that read from `storage` to `view`. Otherwise, some functionality may break in future Solidity versions where this bug is fixed.

Update: *Resolved at commit 369b85b.*

L-17 Successful ERC-20 operations' execution is not enforced

Some ERC-20 operations throughout the codebase rely on the underlying implementation reverting on failure. For example:

- The `WrapperModule` contract uses the plain `IERC20` interface to perform actions like `transferFrom`. However, some tokens do not revert on failed transfers and return

`false` instead, like ZRX. Moreover, one should not assume that `transferFrom` always returns a bool. For instance, USDT never returns `true`.

- When dealing with synths in escrow, tokens can be transferred into the spot market contract (into escrow), transferred out to the trader (out of escrow) or burnt. In all of these scenarios, the returned value is not checked. The synth transfer could have failed, but if instead of reverting it returned `false`, the total escrowed shares' accounting would be wrong. The current `SynthTokenModule` implementation does revert on failure, but it can be upgraded arbitrarily by the synth owner.

Consider enforcing a successful result when calling these functions in order to prevent potential accounting issues in the future, should synth market owners decide to upgrade the synth token implementations, or should governance decide to allow more exotic collaterals to be wrapped. In order to cover all cases, consider leveraging the `SafeERC20` library. Additionally, consider enforcing the checks-effects-interaction pattern when transferring funds into escrow.

Update: *Partially resolved at commit 019b3dd. The instance in `AsyncOrder.sol` was left unresolved.*

L-18 Undocumented and incomplete functionality

According to the team, incorporating the off-chain settlement of Chainlink-reported price orders is not intended. This approach may incorrectly scale the retrieved median and provides only a partial verification of the timestamp's validity.

Since it has been clarified that this feature will not be initially implemented, consider removing it from the codebase to avoid unintentional and unexpected interactions with it.

Update: *Resolved at commit 4af6ab9.*

L-19 `setWrapper` does not limit the `maxWrappableAmount`

The `setWrapper` function of the `WrapperModule` lets the synth's owner set a `maxWrappableAmount` parameter. However, this value should be restricted, since the core protocol has a similar parameter that will definitely be taken into account when wrapping.

Consider limiting the value of `maxWrappableAmount` to be smaller than or equal to `maximumDepositableD18`.

Update: *Acknowledged, not resolved. The Synthetix team stated:*

> *We think it makes sense to allow the maximum wrappable amount to be set independently of the system maximum amount, as the latter could be changed.*

L-20 Fixed fees cannot be zero

The `setCustomTransactorFees` function of the `FeeConfigurationModule` lets the synth owner override fixed fee values for specific actors. However, these fees cannot be set to zero (to avoid charging fixed fees entirely). This is because when fees are retrieved, the use of overridden fees is determined by whether they are greater than zero.

Having the possibility to set fees to zero seems to be a desired outcome, so that privileged fees can be revoked. However, this implies that free trading is not allowed.

If this is the desired behavior, consider explicitly stating it in the docstrings.

Update: *Resolved at commit 8f764ee.*

L-21 Unnecessary conditional price scaling on Pyth node

The `PythNode` oracle node assumes that the `pythData.expo` parameter returned by the price retrieval is negative and can have an arbitrary value.

It seems that this exponent is usually returned as a negative value from the `Pyth` protocol. However, this is not clear from their documentation, and after a brief analysis of their codebase, the `expo` parameter is meant to stay between `-12` and `+12`, effectively allowing a positive `expo` parameter.

If this holds true, then `factor` can never be negative since it will always be greater than or equal to (18 - 12 = 6), and at most (18 + 12 = 30). If the `expo` happens to be positive, it is correct to upscale the price by the sum of `PRECISION` and the exponent. The same holds true if the exponent is negative, where it would be correct to upscale the price by the difference between `PRECISION` and `expo`. Altogether, this implies that the `else` statement of the conditional block will never be executed and should be removed. Notice that the same applies when settling off-chain async orders via `Pyth`.

Consider whether the `else` statement should be removed and whether the code should check for the `expo` parameter to be non-positive.

Update: *Acknowledged, not resolved. The Synthetix team stated:*

> *Supporting any expo value makes the code more robust.*

Notes & Additional Information

N-01 Incorrect event emission parameter

When an async order is settled, the `OrderSettled` event is always emitted.

According to the provided docstrings, the last parameter for this event is the trader's address. However, the `msg.sender` value is being used, which resolves to the keeper's address instead of the trader's, which is not expected to be the same unless a user decides to settle their own trade.

Consider assigning the owner of the async order claim as the `trader` parameter to avoid emitting and returning incorrect information that could hinder off-chain services' ability to track specific events. Alternatively, consider modifying the documentation about the event if this value should indeed represent the keeper's address.

Update: *Resolved at commit ba772ab. The documentation changed so that it reflects that the value being sent is the keeper's address.*

N-02 Unused custom errors

Throughout the codebase, there are several custom errors defined that are unused and could be removed:

- `InsufficientFunds` in the `IAtomicOrderModule`
- `InsufficientAllowance` in the `IAtomicOrderModule` (already defined within `IERC20`)
- `IncorrectExternalNodeInterface` in the `INodeModule` interface
- `InsufficientFunds` in the `IWrapperModule`
- `InsufficientAllowance` in the `IWrapperModule` (already defined within `IERC20`)
- `InvalidPrice` in the `PriceDeviationCircuitBreakerNode`

Consider removing these custom errors if they are indeed unnecessary.

Update: *Resolved at commit 47310fa.*

N-03 Multiple Solidity versions in use

Throughout the codebase, there are different versions of Solidity in use. For example, the `UniswapNode` contract is using version `>=0.8.0`, while the `StalenessCircuitBreakerNode` contract can be compiled with any version greater than `0.8.11` and lower than `0.9.0`. Moreover, contracts are not set to a specific Solidity version, and thus rely on the one set at the project level.

To avoid unexpected behavior, all contracts in the codebase should be compiled with the same Solidity version. Moreover, consider pinning a specific Solidity version that is up to date with the latest fixes.

Update: *Partially resolved at commit 41068c2. Consistency in Solidity versions was improved but a range has been set rather than pinning a specific version.*

N-04 Inconsistent storage slot loading

The method for loading storage slots is inconsistent across the codebase. Sometimes a specific variable is used, while sometimes a hardcoded value is used instead.

To improve the overall quality of the codebase, consider being consistent in how slot locations are saved and used to retrieve the relevant storage.

Update: *Acknowledged, not resolved. The Synthetix team stated:*

> *Constant values are used when a dynamic parameter is not necessary.*

N-05 NFT URI cannot be changed

In the `NftModule`, there is no restricted function to change the `uri` of the related NFT contract. However, due to the off-chain storage of NFT-related information, it may be necessary to update this value at some point in the future, when the old `uri` is no longer accessible.

If deemed necessary, consider implementing a restricted function to update this value.

Update: *Resolved at commit 0699e7c.*

N-06 Unclear re-initialization feature

The `NftModule` contract has an `initialize` function that marks the `initialized` variable of the `Initialized` storage. This means that once the `initialize` function is called, the contract cannot be re-initialized again. On the contrary, the `TokenModule` uses `decimals` to know whether the contract has already been initialized, but nothing prevents the first initialization from setting `decimals == 0` (as well as the other initialization variables). If this were to occur, `TokenModule` could be re-initialized again.

Additionally, the `AssociatedSystemModule` contract only initializes both contracts on the first creation, but any later upgrades to them do not call the `initialize` function again. Regarding unmanaged systems, those are never initialized, and it is not clear whether they are initializable or not (or whether they should be).

Consider documenting whether contracts are meant to be initializable or re-initializable in the docstrings, and if not, consider ensuring that `TokenModule` cannot be re-initialized.

Update: *Partially resolved at commits 29caac17 and 6ae84fc. The team acknowledges that the `initialize` function might be called again under some circumstances.*

N-07 Functions with incorrect visibility

There are some function definitions in the codebase where the specified visibility is incorrect for their use cases. For example, the `initialize` function of the `TokenModule` and the `NftModule` are declared as `public`, but can be `external` instead. Another example is the `acceptMarketOwnership` function of the `SpotMarketFactoryModule`.

If a function is declared as `public` but is never called within the code of the contract or those inheriting from it, consider making it `external`. Consider reviewing the codebase to locate more instances of incorrect visibility - this will improve the overall quality and gas consumption of the code.

Update: *Partially resolved at commit 6ae84fc.*

N-08 Inconsistent calls through inheritance

In the `DecayTokenModule` contract, several calls are routed to the inherited `ERC-20` contract. Sometimes these calls use internal functions, while sometimes they use public ones.

Consider making this behavior consistent by changing the `_transferFrom` call to `transferFrom` instead.

Update: *Resolved at commit c00e684 as part of fixes for C-07 issue.*

N-09 Unused imports

Throughout the codebase some imports are unused and could be removed. Here are some examples:

- Import `AsyncOrderConfiguration` of `IAsyncOrderModule.sol`
- Import `AssociatedSystemsModule` of `FeeConfigurationModule.sol`
- Import `OwnableStorage` of `Proxy.sol`
- Import `Price` of `SpotMarketFactory.sol` along with the `using` statement

Consider checking the entire codebase for unused imports, and removing them to improve its overall clarity and readability.

Update: *Partially resolved at commit 93576ff. The instance in `Proxy.sol` remains unfixed.*

N-10 Non-explicit imports are used

Non-explicit imports can decrease the clarity of the codebase, and may create naming conflicts between locally defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity files or when inheritance chains are long.

Throughout the codebase, global imports are being used. Below is a non-exhaustive list of instances where non-explicit imports are used:

- Line 4 of `IAsyncOrderConfigurationModule.sol`
- Line 4 of `IAsyncOrderModule.sol`
- Line 4 of `IAtomicOrderModule.sol`
- Line 6 of `ISpotMarketFactoryModule.sol`
- Line 4 of `ISynthTokenModule.sol`
- line 4 of `IFeeCollector.sol`
- Line 7 of `AsyncOrderConfigurationModule.sol`
- Line 4 of `AsyncOrderModule.sol`
- Line 8 of `TokenModule.sol`

Following the principle that clearer code is better code, consider using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

Update: *Resolved at commit 5295218.*

N-11 `TODO` comments in the code

Several TODO comments were found in the codebase. These comments should be tracked in the project's issue backlog and resolved before the system is deployed:

- Both `TODO` comments on lines 162 and 165 in `AsyncOrderModule.sol`
- The `TODO` comment on line 165 in `AsyncOrderModule.sol`
- The `TODO` comment on line 299 in `AsyncOrderModule.sol`
- The `TODO` comment on line 136 in `WrapperModule.sol`

During development, having well-described TODO comments will facilitate the process of tracking and solving them. However, these comments can age and important information related to the security of the system could be forgotten by the time it is released to production.

Consider removing all instances of TODO comments and instead tracking the associated tasks in the issues backlog. Alternatively, consider linking each inline instance to the corresponding issues backlog entry.

Update: *Resolved. As of the latest `main` commit 1468027 at the time of writing, the issue is no longer present in the codebase.*

N-12 Lack of indexed parameters

Throughout the codebase, some instances of event parameters could benefit from indexing. For example:

- The `SynthImplementationUpgraded` 's proxy and implementation addresses
- The `SynthPriceDataUpdated` feed ids
- The `MarketOwnerNominated` 's `newOwner` address
- The `MarketOwnerChanged` 's old and new owner addresses
- The `WrapperSet` 's `wrapCollateralType` address
- The `SettlementStrategyAdded` 's parameters in the `IAsyncOrderConfigurationModule` interface
- The `SettlementStrategyUpdated` 's parameters in the `IAsyncOrderConfigurationModule` interface

- The `BatchPriceFeedUpdate`'s parameters in the `IPyth` interface

Consider indexing event parameters to improve off-chain services' ability to search and filter for specific events.

Update: *Partially resolved at commit ec3690a.*

N-13 Interface inconsistencies

Throughout the codebase, there are some mismatches between the interface's definitions and the implementation contracts. Some examples are:

- The `createSynth` function returns a named variable (`synthMarketId`) in the interface, but it does not in the implementation.
- All settling functions within the `IAsyncOrderModule` interface present named return variables, while their implementations do not.
- The `nominateMarketOwner` function has a `marketId` parameter, but the parameter in the corresponding implementation is named `synthMarketId`. The same is true for `acceptMarketOwnership` and `getMarketOwner`.
- The `unwrap` and `wrap` functions of the `IWrapperModule` interface return an unnamed parameter, but the implementations return a named parameter. Moreover, the docstrings state that they return an `amountReturned`, but the returned parameters have different names in the implementations.
- The `buy` and `sell` functions of the `IAtomicOrderModule` interface have parameters with names that differ from the implementation.

In order to improve the quality and readability of the codebase, consider making interfaces consistent with their implementations.

Update: *Resolved at commit 648a809.*

N-14 Gas optimizations are possible

Throughout the codebase, there are several opportunities for gas optimizations. For example:

- When upgrading a synth's implementation, the synth's information is loaded as an associated system, and its implementation value is updated. However, after performing the actual upgrade, the `_setAssociatedSystem` function is called, and will perform three unnecessary `SSTORE`s on all three values (proxy, implementation and kind).

Consider emitting the `AssociatedSystemSet` event directly instead of rewriting these values in storage.

- When upgrading a synth's implementation, `synthId` is calculated twice, once before performing the upgrade and once again when emitting the event.
- When collecting fees, if the `feeCollector` collects 100% of the total fees, `feesToDeposit` will be zero. Consider skipping the `depositToMarketManager` call in this scenario.
- When performing an atomic buy order, the entire snxUSD amount provided by the trader will get deposited through the market manager. Instead of depositing the trading amount (after fees) and the fees in two steps, consider refactoring the logic so that the `collectFees` return value is used to calculate the total amount to deposit in one contract call.
- When issuing atomic orders, the price is read twice from the oracle feed. The first time takes place when calculating the exchange rate internally within the `Price` library, while the second time takes place as one of the parameters when calling `processFees`. Consider refactoring the code so that it is only checked once.
- When a wrapping operation results in positive fees, the `totalFees` amount is withdrawn from the market in snxUSD. If there is no `feeCollector` set, this amount will be redeposited back into the market. Consider checking if there is a `feeCollector` set before withdrawing USD from the market in order to avoid performing both operations.
- When unwrapping a synth, there is no need to pull trader funds before burning them since the market can burn synths without any allowance from the trader.
- When processing prices for oracle nodes that have a list of parent nodes, the parent nodes need to be processed first. When looping through each parent node, the code is performing an external call to the `process` function, instead of calling the internal function directly. Consider calling the internal function in order to save gas, unless this behavior is intended. If that is the case, consider adding docstrings explaining the rationale behind the decision.

Consider resolving these cases, and exploring the codebase for more instances where gas optimizations are possible. When deciding whether to fix these to lower gas consumption, consider whether there are other parts of the codebase that may be affected, and if it would imply changing the security assumptions.

Update: *Partially resolved at commit a1397c5.*

N-15 Typos

There are several typographical errors throughout the codebase:

- In the `INftModule` interface, "wether" should be "whether".
- The `skewScale` parameter's docstrings should say "outstanding" instead of "outsanding".
- In line 57 of the `IAtomicOrderModule` interface, "allownace" should be "allowance".
- In the `readme` file of the oracle manager, "can registered" should be "can be registered".
- In the `readme` file of the spot market, the word "asyncronous" should read "asynchronous". This error is repeated in multiple instances throughout the text.

Consider fixing these typographical errors to improve the quality and readability of the codebase.

Update: *Partially resolved at commit ee2c47e.*

N-16 Hardcoded values

In the `totalSupply` function of the `DecayTokenModule`, `10 ** 18` is used, when `DecimalMath.UNIT` could be used instead. The same occurs in the `calculateUtilizationRateFee` function of the `FeeUtil` contract.

Consider using well-defined constant variables instead of hardcoded numbers to improve the readability of the codebase.

Update: *Resolved at commit ea84b08. The `FeeUtil` contract no longer exists in this commit.*

N-17 Missing or incorrect docstrings

Several docstrings and inline comments throughout the codebase were found to be erroneous and should be fixed. In particular:

- `IDecayTokenModule`'s docstrings state that the formula to calculate decay is `P(1 + r/n)**nt`, but the implementation resolves to `P(1 - r)**t`.
- All storage libraries within the `core-modules` and `oracle-manager` packages lack docstrings.
- In the `ITokenModule` interface's `burn` function, "to" should be "from" instead.

- The functions implementing the `IMarket` interface within the `SpotMarketFactoryModule` lack the `@inheritdoc` instructions.
- The entire `SynthUtil` library lacks docstrings.
- Most of the functions within the `SpotMarketFactory` library lack documentation.
- The `SynthSold` event's description should say "sell trade" instead of "buy trade".
- Line 128 of `UniswapNode` should say "call" instead of "return".
- Most of the functions within the `AsyncOrder` library lack documentation.
- All `validate` and `process` functions in the libraries inside the `oracle-manager/contracts/nodes` directory are poorly documented. The same is true for the functions in the `AsyncOrderConfiguration` library.
- The `buyFeedId` parameter of the `Price.Data` struct is said to be used for calculating the reported debt, but this is not true, since the sell feed is used instead.

Consider fixing the erroneous docstrings, but also writing extensive documentation for any relevant contract and/or function. This should improve the overall quality of the codebase as well as its readability.

Update: *Partially resolved at commit d509eb0. No changes have been applied to the `DecayTokenModule`, the storage libraries of `code-modules` and `oracle-manager` and the `nodes` in `oracle-manager`.*

N-18 Fallback node of `PriceDeviationCircuitBreakerNode` may exceed the price tolerance

In the `PriceDeviationCircuitBreakerNode`, whenever the difference between the primary and comparison prices exceeds the desired tolerance (or if the primary price is zero), provided there is a third parent node configured, the reported price belonging to that third price feed is not checked against exceeding the desired tolerance (compared to the other two).

Consider deciding whether this is the intended design. If this is the case, consider expanding the library docstrings to justify the decision.

Update: *Resolved at commit 44fd145.*

N-19 Misleading timestamp when querying Chainlink with a TWAP interval

When querying prices from a Chainlink oracle node, the `timestamp` value is always set to the latest `updatedAt` value provided by the `latestRoundData` call.

This `timestamp` value will always be the same, regardless of the `twapTimeInterval` value selected. This means that, in the eyes of the caller, prices will differ depending on the `twapTimeInterval` but the `timestamp` will always be the same. The problem is that the `twapTimeInterval` can potentially be arbitrarily large and thus, make the final reported price less accurate than the latest reported price, despite reporting a recent `timestamp`.

Consider deciding whether this behavior is intentional, and including a justification. Otherwise, consider whether a more accurate `timestamp` could be calculated based on the `updatedAt` reported value for the last round, and the `twapTimeInterval` value.

Update: *Resolved at commit d59f693.*

N-20 Misleading function names

Throughout the codebase, several instances were identified where function names are misleading in terms of their expected behavior:

- `isEligibleForCancelation`
- `isValidSettlementStrategy`
- `isValidAmount`
- `isAsyncTransaction`

The common issue with these functions is that the name suggests a boolean return value, which would indicate some kind of state. However, in reality, they are used as enforcement checks that will revert on invalid conditions and go through without return values if the conditions are valid.

In contrast to these functions, the opposite effect occurs within the oracle manager package:

- The `_validateNodeDefinition` function implies that validation will be performed and if unsuccessful, it will revert. However, it returns a boolean indicating the validation state.

- The `validate` function in all the nodes behaves in a similar way, returning a boolean indicating the validation status instead of reverting when an invalid configuration is detected.

Consider renaming these functions to something like `checkAmountValidity` instead of `isValidAmount` in order to clarify the functions' intention, and in the case of the oracle manager's functions, consider renaming `validate` to `isValid` (or something similar).

Update: *Resolved at commit 4efb394.*

N-21 `abs` function should return `uint256`

The `abs` function within the `SettlementStrategy` library returns an `int`, but by definition, the absolute value will always return a non-negative value.

Consider changing the return value to `uint256` and casting both `x` and `-x` to `uint` since it is safe to do so, and saving some gas by avoiding the unnecessary use of `SafeCastI256`, instead of casting to `uint` everywhere else in the code.

Update: *Resolved at commit 72bda48.*

N-22 `marketId`'s existence is not always verified

The `AsyncOrderModule` heavily relies on the existence of `marketId` but does not always verify its existence. Even if we did not find security issues associated with this behavior, this could be problematic in future upgrades.

Determine if it is essential to validate its existence in functions that do not perform such verification.

Update: *Resolved. This is not an issue because the verification was being done. The team stated:*

> *`_performClaimValidityCheck` should be enough to ensure it's not an invalid market.*

N-23 `settlePythOrder` does not verify that `msg.value` is sufficient

The `settlePythOrder` function of the `AsyncOrderModule` contract is forwarding `msg.value` to the Pyth verifier in order to pay fees.

As stated in Pyth's docstrings, this must be at least the amount returned by the `getUpdateFee` function.

Consider integrating a check for this value instead of letting the function revert if `msg.value` is not sufficient.

Update: *Acknowledged, not resolved. The Synthetix team stated:*

> *The caller of this function should interact with Pyth's contract directly to retrieve the quote, per documentation.*

N-24 Default value is used with a meaningful purpose

The null value of the `Type` enum in the `Transaction` library is used with a meaningful purpose. The same occurs in the settlement strategies.

Consider defining `NULL` before `BUY` and avoid relying on default values to express non-null meanings.

Update: *Resolved at commit e64bc0b.*

N-25 Unmanaged systems can be expected to be anything

The `expectKind` function of the `AssociatedSystem` library will not revert if the `actualKind` read from storage is an unmanaged system, and the parameter `kind` is any other value.

Unmanaged systems can be expected to be anything without causing the `expectKind` function to revert. Consider deciding whether the `expectKind` function should revert if an unmanaged system is expected to be anything other than `KIND_UNMANAGED`.

Update: *Acknowledged, not resolved. The Synthetix team stated:*

> *"Unmanaged" associated systems are meant to support any external contract and only track the external address (e.g., the address for integrated cross-chain bridges).*

N-26 Deniers are stored inefficiently

The `deniers` array of the `FeatureFlag.Data` struct is not an instance of `SetUtil.AddressSet` (like `permissionedAddresses`). The objective is to efficiently store a set of addresses which are configured to be deniers. Because of this, checking whether an address is in the `deniers` array requires looping through the entire array. On the contrary, the `SetUtil.AddressSet` has specific modular functions that do not require writing extra code to handle the same feature. At the same time, this complicates the functionality of changing elements in the array, or retrieving its values.

Consider changing the `deniers` array to a `SetUtil.AddressSet` instead.

Update: *Acknowledged, not resolved. The Synthetix team stated:*

> *While it is true that it would be more efficient for deniers to be stored in an `AddressSet` for the purposes of later retrieval, in an effort to keep the code as simple as possible and compliant with our deployment standards, we have decided to keep the structure as a simple list. In general, we expect few addresses to be able to call `setFeatureFlagDenyAll` and this call is rarely executed.*

N-27 Fallback node of `StalenessCircuitBreakerNode` is not checked for staleness

In the `StalenessCircuitBreakerNode`, whenever the first parent is stale and the second parent is set and not returning a price of zero, the second parent's output is returned. However, this output is never checked to be a non-stale price.

Consider deciding whether this is the intended design, and expanding the docstrings of the contract to further document this behavior.

Update: *Resolved at commit e814742.*

N-28 Unclear custom error parameter

The `DeviationToleranceExceeded` custom error of the `PriceDeviationCircuitBreakerNode` reports the deviation as `type(int256).max` or `difference / primaryPrice`, depending on whether the `primaryPrice` is zero or not.

The conditions under which this error is triggered are:

- If the `primaryPrice` is zero and the parent's length is smaller than or equal to 2. In this case, a third parent is not set as a fallback, and the price is zero. As such, the error has nothing to do with the fact that the tolerance has been exceeded, resulting in a confusing custom error parameter reported. Moreover, it would be triggered with `type(int256).max` which does not add any meaningful value to the context.
- If the `primaryPrice` is not zero, and the tolerance has been effectively exceeded. In this case, the error correctly signals what is happening, but may indicate a wrong value since `difference / primaryPrice` will potentially truncate the result down. What the error should be doing is upscaling the `difference`, similarly to what is done when comparing it against the tolerance value.

Consider refactoring this part of the codebase to make consistent use of explicit custom errors and report meaningful error messages and values.

Update: *Resolved at commit 3fb38e2.*

N-29 Unnamed return parameters

There are multiple instances where returned parameters are not named. For example:

- All the functions in the `SpotMarketFactoryModule`
- All the functions in the `SynthUtil` library
- All the functions in the `AsyncOrder` library
- Most functions in the `AsyncOrderModule` and the `getAsyncOrderClaim` function in its corresponding interface
- Most functions in the `NodeModule` and its corresponding interface

Consider using named return parameters to improve the explicitness and readability of the codebase.

Update: *Partially resolved at commit 648a809. The instances in the `SynthUtil` library remain unresolved.*

N-30 Inconsistent use of implicit int and uint types

Throughout the codebase, there is inconsistent use of `uint` and `int` vs the fully qualified `uint256` and `int256` data types. Given that the system uses several specifically-sized variables such as `uint64` or `int128`, the use of plain `uint` and `int` can be unclear and may hinder readability.

Some examples are:

- All `uint256` instances within the `_settleOrder` function
- All `uint256` and `int256` instances within the FeeConfiguration storage library
- All `int256` instances within the `ReducerNode` `quickSort` function

To favor explicitness, consider changing all instances of `uint` and `int` to `uint256` and `int256` respectively.

Update: *Resolved at commit 019ea49.*

Conclusion

Several critical and high-severity vulnerabilities have been found, amongst other minor issues. The team provided swift answers and sufficient documentation to support the audit. However, given the number and significance of issues identified in the spot market, it is probable that further issues may surface as the team implements solutions. Fixing all of the `spot-market` issues will require significant refactoring, which will impact other packages, including the `core` protocol package. Consequently, the review process for the fixes will become more complex, and there is a risk of overlooking any potential repercussions in other areas of the codebase. Fortunately, the issues with the `core-modules` and `oracle-manager` can be addressed without any additional iterations.

Appendix

Monitoring Recommendations

While audits help in identifying potential security risks, the Synthetix team is encouraged to incorporate automated monitoring of on-chain contract and mempool activity into their operations on all networks in use. Ongoing monitoring of deployed contracts helps identify potential threats and issues affecting the production environment. In this case, it may also provide useful information about how the system is used or misused. Consider monitoring the following items:

- Unrealistic slippage settings for async orders: this may suggest malicious behaviors that prevent orders from being settled.
- Transfer of synths, snxUSD or any collateral directly into the contracts: this may suggest manipulation attempts of `balanceOf(address(this))`.
- High-frequency trading addresses: This may suggest automated traders but also draining (or similar) attacks.
- Synth owner actions: if a synth owner becomes malicious, there are several ways by which investors and users can have their funds lost or stolen. Any change in settlement strategies, fees, price feeds or synth token implementation upgrades should be monitored.
- Oracle interactions that start reverting: this may suggest an ongoing oracle price manipulation attack or oracle failure.