OpenZeppelin | security

Synthetix V3 Phase 1 Audit

SYNTHETIX

February 21, 2023

This security assessment was prepared by OpenZeppelin.

Table of Contents

Summary

Type	DeFi
Timeline	From 2022-01-03 To 2022-02-02
Languages	Solidity

Total Issues	46 (36 resolved, 2 partially resolved)
Critical Severity Issues	1 (1 resolved)
High Severity Issues	5 (5 resolved)
Medium Severity Issues	5 (5 resolved)
Low Severity Issues	17 (12 resolved)
Notes & Additional Information	18 (13 resolved, 2 partially resolved)

Scope

We audited the `Synthetixio/synthetix-v3` repository at the `9f3c2c82be411278b9349fca1acd6e5c0685a77d` commit.

In scope were the following contracts:

```
markets/legacy-market/
├── contracts
│   ├── InitialModuleBundle.sol
│   ├── LegacyMarket.sol
│   ├── Proxy.sol
│   └── interfaces
│       ├── ILegacyMarket.sol
│       └── external
│           ├── IRewardEscrowV2.sol
│           ├── ISynthetix.sol
│           └── IV3CoreProxy.sol
protocol/synthetix/
└── contracts
    ├── Proxy.sol
    ├── interfaces
    │   ├── IAccountModule.sol
    │   ├── IAccountTokenModule.sol
    │   ├── IAssociateDebtModule.sol
    │   ├── ICollateralConfigurationModule.sol
    │   ├── ICollateralModule.sol
    │   ├── IIssueUSDModule.sol
    │   ├── ILiquidationModule.sol
    │   ├── IMarketCollateralModule.sol
    │   ├── IMarketManagerModule.sol
    │   ├── IMulticallModule.sol
    │   ├── IPoolConfigurationModule.sol
    │   ├── IPoolModule.sol
    │   ├── IRewardsManagerModule.sol
    │   ├── IUSDTokenModule.sol
    │   ├── IUtilsModule.sol
    │   ├── IVaultModule.sol
    │   └── external
    │       ├── IAggregatorV3Interface.sol
    │       ├── IAny2EVMMessageReceiverInterface.sol
    │       ├── IEVM2AnySubscriptionOnRampRouterInterface.sol
    │       ├── IMarket.sol
    │       └── IRewardDistributor.sol
    ├── modules
    │   ├── InitialModuleBundle.sol
    │   ├── account
    │   │   └── AccountTokenModule.sol
    │   ├── associated-systems
```

```
    │   │   └── AssociatedSystemsModule.sol
    │   ├── common
    │   │   ├── OwnerModule.sol
    │   │   └── UpgradeModule.sol
    │   ├── core
    │   │   ├── AccountModule.sol
    │   │   ├── AssociateDebtModule.sol
    │   │   ├── CollateralConfigurationModule.sol
    │   │   ├── CollateralModule.sol
    │   │   ├── FeatureFlagModule.sol
    │   │   ├── IssueUSDModule.sol
    │   │   ├── LiquidationModule.sol
    │   │   ├── MarketCollateralModule.sol
    │   │   ├── MarketManagerModule.sol
    │   │   ├── MulticallModule.sol
    │   │   ├── Poc.sol
    │   │   ├── PoolConfigurationModule.sol
    │   │   ├── PoolModule.sol
    │   │   ├── RewardsManagerModule.sol
    │   │   ├── UtilsModule.sol
    │   │   └── VaultModule.sol
    │   └── usd
    │       └── USDTokenModule.sol
    └── storage
        ├── Account.sol
        ├── AccountRBAC.sol
        ├── Collateral.sol
        ├── CollateralConfiguration.sol
        ├── CollateralLock.sol
        ├── Distribution.sol
        ├── DistributionActor.sol
        ├── Market.sol
        ├── MarketConfiguration.sol
        ├── MarketCreator.sol
        ├── MarketPoolInfo.sol
        ├── OracleManager.sol
        ├── Pool.sol
        ├── RewardDistribution.sol
        ├── RewardDistributionClaimStatus.sol
        ├── ScalableMapping.sol
        ├── SystemPoolConfiguration.sol
        ├── Vault.sol
        └── VaultEpoch.sol
```

System overview

This audit covers major components of the new Synthetix protocol, v3, which introduces significant improvements over the previous version, known as v2x. The protocol enables users

to place collateral and assign it to pools for exposure to fluctuating debt, and also provides them with the ability to borrow the stablecoin snxUSD against their collateral.

The fluctuating debt comes from markets that make use of the system credit capacity. Each pool can support one or multiple markets, thereby offering credit for use. Markets are the final end entity of the system - they use credit given by the system to let users operate with synths in each specific market. Synths are tokenized representations of other assets. Each market specifies its own logic, whether it is a spot market or any other financial instrument that makes use of them.

The system features three novel components, described below.

ScalableMapping data structure

The `ScalableMapping` data structure improves normal mappings when trying to update all user balances at once. In a normal mapping, it takes O(n) to update each balance by multiplying it by a certain scalar, while in this structure it takes O(1).

The key for the mapping is a `bytes32` ID so that it can be abstracted from addresses, and thus represent any other entity, such as a vault or a pool. It is very helpful when trying to socialize debt or liquidated collateral across all participants at once.

It keeps track of a number of shares per actor involved, along with a `scaleModifier` so that individual values can be calculated as `(1+scaleModifier) * actor_shares`.

Distribution data structure

The `Distribution` data structure enables the tracking of a global value distributed among a set of actors (DistributionActor). Total shares can be scaled with a valuePerShare multiplier and individual actor value can be calculated as their amount of shares times this multiplier.

It contains the following fields:

- The total number of shares within the `Distribution`, where each share usually represents one USD worth of collateral.
- A value per share, which usually represents the USD amount of debt per USD worth of collateral.
- A mapping linking actor IDs with a `DistributionActor` structure, which holds information about how many shares each actor holds and the last value per share registered.

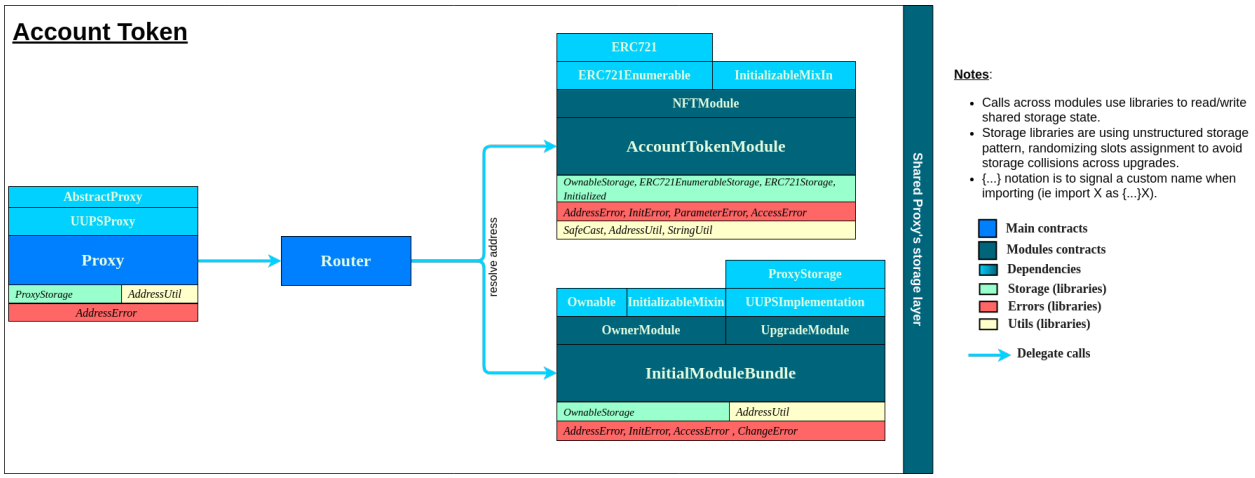
Changes for each individual actor can be tracked thanks to last value per share being stored on each interaction.

Router pattern

Complex systems necessitate many contracts, and require intercommunication. Oftentimes, when calls between contracts occur, they must know whether the call is coming from an official "system" contract.

Additionally, multiple inheritances can produce unexpected storage layouts, and it becomes hard to detect invalid mutations, and therefore tooling is sometimes needed.

The router pattern utilizes a static routing table that is generated through specialized software, allowing a UUPS proxy to forward requests to the appropriate module through the use of `delegatecall`. Inter-module communication is straightforward as each module simply needs to import the necessary storage library and access storage. To avoid storage collisions, randomized storage slots are employed. An illustration of this design can be viewed in the diagram below, depicting a module responsible for minting an NFT token associated with an account:



System architecture

As with the router pattern, all code is split across different modules, which import storage libraries to share access to storage. Listed below are summaries of the relevant core modules.

Associated systems module

The core protocol is accessed through a main router, and all components can freely communicate amongst each other. However, sometimes interaction with external systems is necessary, and that is what this module aims to solve.

For instance, `snxUSD` minting and burning is performed by registering the token as an ERC-20 type associated system, so all interactions with it are managed through the core system. Associated systems can be ERC-20, ERC-721 or unmanaged. An example of an unmanaged system would be interactions with Chainlink's Cross-Chain Interoperability Protocol (CCIP), while an example of an ERC-721 associated system is the account token module.

Account token module

Accounts are represented by NFTs, which can be minted on account creation and transferred out to a third party thanks to this module. Each account has a unique id represented by the NFT id. NFTs can be transferred, which transfers ownership of the account.

Initial module bundle

This module helps kickstart system deployment by combining ownership and upgradability management first, before the rest of the system is deployed.

Owner module

This module takes care of system ownership initialization and management.

Upgrade module

The upgrade module allows the system to be upgraded either fully or partially. It features a protection mechanism to ensure the upgraded implementation is actually upgradable, and protects the system from becoming bricked.

USD token module

The USD token module allows minting and burning of `snxUSD` by authorized parties, such as the system owner or Chainlink's CCIP in order to transfer assets cross-chain. This last feature is not yet available, but will be enabled in the future.

Account module

The account module enables account management, which includes fine-grained role-based access to perform actions from other addresses such as deposits, rewards claiming or withdrawals. Accounts storage also handles the collateral owned by an account and deposited in the protocol, the collateral locked for a certain amount of time, and the pools to which that collateral is delegated (to provide them with credit capacity).

Associate debt module

Sometimes markets need to associate a specific amount of debt to an individual account due to interactions. The associate debt module provides the proper entry points to do so, adjusting the whole debt distribution and ensuring minimum collateralization ratios are met.

Collateral configuration module

The system owner can decide which assets can be used as collateral within the protocol, and this is the entry point to enable, configure and disable them.

Collateral module

The collateral module is the first entry point for end-users looking to deposit their assets within the system. Before being able to participate in the debt distribution chain, they need to deposit

their assets through this module. It also allows withdrawals and collateral lock creation and cleanup.

Feature flag module

In order to smoothly deploy the system, some features are going to be restricted to the general public. The feature flag module ensures that only authorized users can access specific features.

Update: *The Synthetix team provided the following statement as an update:*

> *The functionality of this module has been expanded slightly to allow for CCs to independently disable certain functions of the system without having to go through the multisig. Feature flags are now used on almost every non-system owner function as of the latest code as a safety precaution.*

Issue USD module

The issue USD module is dedicated to borrowing and repaying `snxUSD`, provided there is collateral to borrow against.

Liquidation module

Each collateral type has a certain liquidation ratio configured that must be respected at all times. If an individual position breaks this ratio, it can be liquidated in exchange for a reward through this liquidation module.

If the entire collateral and debt held in a specific vault is below the liquidation threshold, then individual liquidations are not allowed and a full (or partial) vault liquidation must be performed instead.

Market collateral module

Markets get access to a certain credit capacity thanks to pools delegating credit to them. However, if they wanted to boost their credit, direct collateral deposits into a market can be performed through this market collateral module.

Update: *The Synthetix team provided the following statement as an update:*

> *Market collateral module is a feature that will likely only be available to very few approved markets that go through a governance process, and there is a limit to how much can be deposited. Otherwise, markets are genreally not allowed to deposit collateral as it can be used maliciously given that `snxUSD` can be withdrawn at a 1:1 ratio with no consequences.*

Market manager module

The market manager module is the main entry point for market managers to deposit and withdraw USD to support their internal operations, being supported by the delegated credit capacity from pools that support them.

Depositing USD equates to burning `snxUSD`, while withdrawing means further borrowing of `snxUSD` by minting it.

Multicall module

When users need to perform several actions in a row, the multicall module can be leveraged to batch multiple actions into one transaction.

Pool configuration module

The system owner can use the pool configuration module to mark some pools as approved, and to indicate a preferred pool. This does not mean they get special treatment, but it is intended to make them stand out as approved by the Synthetix team.

Pool module

The pool module facilitates pool management, which includes creation, ownership transfer, and configuration. Within this configuration, a pool owner can decide which markets to support, and in which proportions.

Rewards manager module

Different parties can decide to reward users by distributing tokens to them. The rewards manager module allows reward distribution creation, management, and claiming.

Utils module

The utils module provides some helper functions related to the CCIP integration and oracle management.

Vault module

Once a user has deposited collateral into the system, they need to interact with the vault module in order to delegate it to specific pools, depending on their deposited collateral type.

Markets

Markets are independent entities that conform to the `IMarket` interface. Pools aggregate different collateral types from multiple users in order to provide credit capacity to these markets. It is up to the pool's manager to decide how much credit capacity is provided to each market using a weights-based system.

Markets can mint `snxUSD` against their delegated credit capacity in order to provide all kinds of services, such as spot or perpetual futures trading. A spot market would be comprised of multiple markets within the same contracts, for instance, providing access to synthetic assets (synths) being collateralized by the delegated credit capacity from the pools.

Within the audited codebase is the Legacy market, which will be the first deployed market from the Synthetix team.

Legacy market

As the first deployed market within the V3 ecosystem, the goals of the Legacy market are threefold:

- Migrate v2x positions into the new v3 system. In order to accomplish this, both collateral and debt will be brought over to v3 by creating a new account, depositing the collateral, delegating it into the preferred pool, and finally associating the v2x debt to it. For convenience, the account is owned by the `LegacyMarket` during creation, and then transferred to stakers' addresses.
- Convert old `sUSD` stablecoin holdings into the new v3 `snxUSD` token. In order to do this, the `LegacyMarket` needs to have received enough credit capacity to support it.
- Act as a v2x large staker for v3 users who want to get exposure to v2x debt. Any pool delegating collateral into this market will get exposure to v2x debt by allowing the conversion from `sUSD` and `snxUSD`.

Notice that while v2x accepts SNX and ETH positions, the `LegacyMarket` allows migrations only of SNX positions.

Debt distribution chain

The debt distribution chain is a two-way path for funds to flow between individual accounts and markets. There are different actors involved on it:

- **Markets:** These entities receive credit capacity from pools in order to carry on different endeavors. Some applications include spot market trading, perpetual futures, etc. Debt will be borrowed and repaid by these markets constantly, counting on the delegated collateral from pools that support them. Markets need to conform to the `IMarket` interface in order to enable communication with the core system. Any debt reported by markets will be passed down to pools.
- **Pools:** Pool owners will aggregate collateral delegated by multiple users and decide which markets they will support. They can specify a set of weights so that credit capacity is distributed asymmetrically to any market they decide to support.
- **Vaults:** Pools may get multiple collateral types delegated to them. Each collateral type will be held in a different vault.
- **VaultEpochs:** When a vault's entire holdings are below the liquidation ratio, they can be liquidated. When this happens, a new epoch is created with fresh storage, so that new delegated collateral can be held in them.
- **Individual accounts:** These are managed by end users who deposit collateral and decide which pools they want to support by delegating their collateral to them. When delegating collateral, depending on the asset type, they will end up in different vaults.

In order to appropriately link all these actors, the `Distribution` data structure plays a critical role:

- `poolsDebtDistribution` features pools as actors. The number of shares within this Distribution represents how much USD worth of collateral they contain, while the `valuePerShare` indicates how much debt in USD terms they have accrued per every USD of collateral. When markets report debt, it gets passed down to pools depending on the configured weights.
- `vaultsDebtDistribution` features vaults as actors. The number of shares within this Distribution represents how much USD worth of collateral each vault holds, while the

`valuePerShare` indicates how much debt in USD terms they have accrued per each USD worth of collateral. When debt is assigned to a pool, it gets distributed across the different vaults that they are composed of.

- `accountsDebtDistribution` features individual accounts as actors. Each share represents a unit of delegated collateral by an individual account, while the `valuePerShare` indicates how much debt per unit of collateral is passed down to the individual user. Once debt reaches this level, it gets consolidated on every user interaction so that each user knows exactly how much debt they have accrued in `snxUSD` terms.
- The final data structure needed is the ScalableMapping `collateralAmounts`, used to track how much collateral has been delegated on each vault. When a liquidation takes place, collateral gets socialized by distributing it equally among all participants.

Apart from receiving debt from Markets, users can borrow `snxUSD` directly against their collateral, as long as they respect the configured collateral ratio for that collateral type.

As synthetic asset prices fluctuate, so will each user's share of debt, so they need to ensure the target collateral ratio is always respected. In contrast to the previous version of the protocol (v2x), users unwilling to be exposed to fluctuating debt can decide to delegate their collateral to a special pool with ID = 0, causing their debt levels to remain constant.

Markets can boost their credit capacity by depositing their own collateral directly. Having access to certain credit capacity gives them the ability to mint synthetic assets for users to trade free of slippage.

Distributing market debt among pools

Different pools might provide different levels of credit capacity, and they can configure a maximum level of debt they are willing to accept from markets.

The algorithm that determines how debt is passed down to pools tries to first fill the pools with the least debt tolerance. Once a pool has reached its maximum amount of debt per USD of collateral, it gets disconnected until the debt is repaid.

Pools are organized in a Heap data structure with their debt tolerances assigned as negative values, so that it is straightforward to pull the smallest pool out of them.

Privileged roles

The following is an overview of the privileged roles within the system, and their powers:

- System owner can:
  - Initialize or upgrade any associated system
  - Manage the allow list of the feature flag module
  - Upgrade and initialize every upgradable module
  - Configure collateral types within the system
  - Configure the maximum depositable collateral for a market
  - Set preferred pools, or add and remove approved pools
  - Set a system-wide minimum liquidity ratio
  - Register the CCIP integration and configure an oracle manager
  - Mint and burn arbitrary amounts of `snxUSD` on behalf of any user, and set allowance for any spender of any account
  - Transfer ownership of the system
- FeatureFlag-approved addresses can access experimental features:
  - Market creation
  - Pool creation
- CCIP from Chainlink can:
  - Mint and burn arbitrary amounts of `snxUSD` on behalf of any user
- Pool owners can:
  - Configure pools
  - Register reward distributors
  - Remove reward distributors
  - Transfer ownership of the pool
- Markets can:
  - Assign debt directly to specific accounts.
  - Call `depositMarketCollateral` and `withdrawMarketCollateral` to directly deposit/withdraw collateral to increase/decrease their capacity.
  - Call `depositMarketUsd` and `withdrawMarketUsd` functions to burn and mint `snxUSD` manipulating their debt up to their capacity and limits.

Trust assumptions

In order for the protocol to work as intended, the system relies on the following assumptions:

- Collateral management does not handle tokens that might subtract a fee from the transfer. This is taken into account at any moment when considering which collateral should be allowed in the system.
- It is always assumed at any moment that `liquidationRatio < issuanceRatio`. If this does not hold, the `delegateCollateral` function might suffer from a security breach due to a failure to check the correct ratios.
- When using collateral locks, the amount locked is always assumed to be accounted for in the `availableAmount`, otherwise the general accounting when cleaning expired locks will break.
- Addresses of several components are set to non-malicious contracts and non-malicious EOAs. An example is the owner of `LegacyMarket`, which can drain its SNX holdings by setting the `v3SystemAddress` to a malicious address when calling `setSystemAddresses` to get infinite SNX approval.
- Users with `ADMIN` role in the `AccountModule` can effectively remove admin rights from other addresses with the same role. The trust assumption is that admins will not become malicious.

Findings

Here we present our findings.

Critical Severity

C-01 Third-party permissions are not revoked on account transfer

An account inside the system is represented by an ERC-721 token. Its ID is stored under the `Account` storage struct in the system, officially representing the user account inside the protocol.

One of the features of the `Account` data structure is a fine-grained access control for segregated roles within the system. The `owner` of an account can give `ADMIN`, `DELEGATE`, `WITHDRAW`, `MINT` and `REWARDS` permissions to specific wallets so that they can act on the owner's behalf. This is done with the `grantPermission` and `revokePermission` functions.

Accounts are abstracted away from user addresses, where the wallet of the owner is just another parameter inside the `AccountRBAC` struct, member of the `Account` one. Thanks to this, transferring ownership of an account inside the system is simple, and is done by just transferring the `ERC-721` token to the recipient of the account transfer.

When an account's `ERC-721` token is transferred to another wallet, a `_postTransfer` hook is called. This hook calls the `AccountModule` contract back to notify the system about the account transfer and do internal accounting to remove permissions from the old allowed addresses, and set the owner to the `to` recipient of the account transfer.

However, the part where the permissions set by the old owner are removed is incorrect. The function goes as follows:

```
function notifyAccountTransfer(address to, uint128 accountId) external override {
        _onlyAccountToken();

        Account.Data storage account = Account.load(accountId);

        account.rbac.revokeAllPermissions(account.rbac.owner);
        account.rbac.setOwner(to);
}
```

The `revokeAllPermissions` function will remove all permissions relative to the `account.rbac.owner` but will not remove any permissions of all other wallets that the old

owner might have given power to. This means that if the old owner previously set another one of their wallets as `ADMIN`, after the token transfer that wallet will still be an `ADMIN` under the new account ownership. As a result, a user that receives an account can be tricked by old addresses having full adminship over their account.

The function body should look more like this instead:

```
function notifyAccountTransfer(address to, uint128 accountId) external override {
        _onlyAccountToken();

        Account.Data storage account = Account.load(accountId);
        address[] memory permissionedAddresses =
account.rbac.permissionAddresses.values();

        for(uint i = 0; i<permissionedAddresses.length; i++) {
           account.rbac.revokeAllPermissions(permissionedAddresses[i]);
        }
        account.rbac.setOwner(to);
}
```

Consider refactoring this function to correctly reset all privileged wallets within an account when transferring the ERC-721 token to a new user.

Update: *Resolved at commit [9edb194](9edb194).*

High Severity

H-01 Locked collateral can be withdrawn

The documentation for the `createLock` function of the `CollateralModule` states that it should prevent withdrawals and not affect collateral delegation.

However, the locked collateral can be withdrawn since there is no check enforced within the withdrawal flow.

The `createLock` function is called inside the `LegacyMarket._migrate` function, which is needed to migrate v2 vestings to v3 contracts. The way vestings are converted from v2 to v3 is by transforming them into locks with expiration times. Vestings are stored in the RewardEscrowV2 contract.

If locked amounts are instantly withdrawable, this means that if a user migrates to v3, their vesting position will go on a collateral lock, thus being instantly withdrawable and bypassing the vesting schedule and expiration time.

Consider modifying the withdrawal flow in order to ensure the requested `tokenAmount` does not exceed the available collateral that is unlocked.

Update: *Resolved in pull request #1338 at commit 2f82bb4 and pull request #1347 at commit 5838609. The largest amount between the assigned and locked collaterals is now subtracted from the deposited amount, resulting in the total amount that is not eligible for withdrawal. The transaction will revert if anyone attempts to withdraw a larger amount than this. The reason behind using the largest amount of the two is that locked collateral should not affect delegation, which means that locked collateral can also be delegated, but not withdrawn.*

H-02 Deposited collateral can be stolen

The `deposit` function of the `CollateralModule` contract is meant to let users deposit collateral tokens into a specific account, indicated by the `accountId` parameter.

However, the `accountId` is never checked to exist, leading to users potentially depositing collateral into non-existent accounts.

If this happens, the collateral will be sitting idle until somebody mints a new account matching that specific `accountId` and steals that stray collateral. Unfortunately, this can be done by anyone that is waiting for such a mistake to happen by either front-running or back-running the erroneous deposit transaction.

In order to avoid this issue, consider enforcing the existence of `accountId` in the `deposit` function or implementing a `deposit` permission to ensure these types of errors are avoided.

Notice that since account and pool creations depend on an `id` passed as an input parameter and are susceptible to front-running, this does not completely solve the front-running problem. A user depositing into a non-existent account can be front run by an account creation transaction, and by the time the deposit transaction is validated, the account actually already exists. Completely solving this problem will require changing the way in which `accountId`s are generated, switching from a user input mechanism to an internal auto-incrementing one.

Update: *Resolved at commit 72b668a. The Synthetix team acknowledged the front-running opportunity and stated:*

> *The front-running concern here should be mitigated through a front-end confirmation that the user is not staging a deposit to a non-existent account.*

H-03 Removing reward distributor does not reset the distribution

The `removeRewardsDistributor` function of the `RewardsManagerModule` is meant to remove a specific `distributor` and its distribution from the list of registered rewards distributions. What the function does is evaluate the `rewardId` of the distribution and remove it from the list of allowed distribution `rewardIds` for a specific collateral and pool.

However, the reward distribution itself is never reset, so once the function executes, `start`, `duration`, `lastUpdate` and `scheduledValueD18` will remain the same as their original values.

On the other hand, a user that starts to delegate collateral to a specific pool is also entitled to the corresponding registered rewards distribution at that moment. This is because the `updateRewards` function is called when delegating, setting a `lastRewardPerShare` value in all the reward distributions for that specific user. The call to `updateRewards` before assigning any delegated collateral makes it possible for the user to only benefit from the rewards from the moment it delegates (by setting `lastRewardPerShare`) and onwards, which avoids the scenario in which the user can claim rewards matured in the past when no

collateral was delegated. Notice that if a distributor has been removed, the corresponding reward distribution will not be updated in here, since the corresponding `rewardId` is not present in the list of `rewardIds`.

Altogether, this means that a `distributor` who was previously removed and later comes back (and is re-added) can trigger the following scenario:

- A distributor is removed and the distribution is left with its values, with a valid `start`, `duration`, `lastUpdate` and `scheduledValueD18`.
- A transaction to set a previously removed distributor is sent to the mempool.
- An attacker that has never delegated collateral sees such transaction and front runs it by delegating some collateral and entitling themselves to all the reward distributions in that collateral and pool. Notice that at this moment, the `distributor` and its distribution are not present in the `rewardIds` list, so the attacker will not have a `lastRewardPerShare` set for the removed distributor's distribution, and its value will be 0.
- The front-run transaction gets executed, so now the distributor is back and the `rewardId` is again in the `rewardIds` list.
- The attacker can now back-run, claiming all the rewards from the old `distributor's` distribution, maximizing their profit given the fact that `lastRewardPerShare` is 0 in this case.

If leaving the old distribution untouched when removing a distributor is the intended design, consider taking into account the aforementioned attack scenario and evaluating whether the system should avoid it. If this is not the case, consider resetting the active distribution values when removing a distributor.

Update: *Resolved in pull request #1439 at commit 05ae8d8 and at commit 749c15a. The Synthetix team decided to disallow re-adding reward distributors once they are removed. In addition to this, any ongoing distribution is stopped and all values are reset to zero. Note that unclaimed rewards can still be claimed from removed distributors.*

H-04 USD can be minted at no cost

The `IssueUSDModule` is supposed to mint `snxUSD` tokens to `msg.sender` once the caller has specified a `poolId` and a `collateralType` to support the debt emission.

However, there is no verification of whether the specified `poolId` actually exists. In the case of a non-existent pool, both the debt and collateral value would be zero. The `collateralType` is not enforced to be enabled either, but at some point in the function

execution, the collateral configuration of such `collateralType` is used to fetch the price from the oracle. If this price is actually retrieved without reverting, regardless of the reported price, the collateral value in USD will be 0, since there are no collateral shares in a non-existent pool.

If a user calls the `mintUsd` function passing a non-existing pool id, the only way to prevent free unlimited USD minting is to hope that the transaction will revert for either of the following reasons:

- The `issuanceRatio` check is not satisfied.
- The oracle fails in retrieving the price.

A `collateralType` that is uninitialized has its `issuanceRatioD18` variable set to 0, so since `newDebt > 0` and `collateralValue == 0`, the calculated collateral ratio equals 0, which avoids reverting with `InsufficientCollateralRatio`. Another way in which the `issuanceRatio` check can be satisfied is if the `collateralType` configuration explicitly sets the ratio to 0, no matter if the collateral type is enabled or not.

In the second case, the transaction would revert if the oracle failed while fetching the price. The oracle manager package is out of scope for the current auditing phase, and it seems to revert when trying to retreive the price retrieval for non-configured node ids. However it seems that anybody can register a node for a collateral type. Since this is out of scope, we cannot assess whether this would make an attack possible, even in the case where the `collateralType` is not initialized in the collateral configurations of the system.

Moreover, if a collateral was enabled and configured in the past (including an oracle node for reporting its price) but has been recently disabled, it could be used to mint unlimited `snxUSD` tokens, since the price would be retrievable.

Although this may not represent an issue, the `burnUsd` function also fails to verify the existence of `poolId`.

To avoid these edge cases and rely on the transaction to revert, consider enforcing that the pool exists, that the specified `collateralType` is enabled or has a positive issuance ratio, and that `accountId` has a non-zero amount of collateral shares before allowing the minting of new debt.

Finally, consider reviewing all the places in which a `poolId` is used and ensuring that the pool's existence is being checked, and as a result avoiding relying on outside checks to eventually revert.

Update: *Resolved at commits 7d21c08 and fa36824. When trying to mint USD, it is now enforced that the collateral type is enabled and its value is strictly positive before checking the collateralization ratio.*

H-05 USD can be minted using disabled collateral

Each collateral that is configured in the system has a `depositingEnabled` boolean parameter. In case there is an issue with the collateral asset, the system can flag the `depositingEnabled` parameter as `false`.

This will prevent users from using such collateral in the `deposit` and `delegateCollateral` functions. Notice that in the `delegateCollateral` call, whether the collateral is enabled or not, there is only a check as to whether the user is increasing their delegation, while reducing the delegation is not prohibited (to eventually also withdraw a disabled collateral). This is meant to let users exit from their positions corresponding to a disabled collateral.

Whether a certain `collateralType` is disabled or not, this does not affect liquidations, nor burning USD, which is aligned with the intention of exclusively restricting a disabled `collateralType` from reducing position sizes and debt exposures through it. However, the `mintUsd` function of the `IssueUSDModule` does not prevent users from using their positions in a disabled `collateralType` to increase their debt exposure by minting more USD.

In the scenario in which a `collateralType` is disabled due to being too volatile, falling in price or just maliciously behaving, users can still increase their debt by using such collateral in the `mintUSD` function, even if the system disables deposits and delegations.

Consider whether this is an issue with the current intention and design and whether `mintUsd` should effectively check for the collateral to not be disabled before minting more USD.

Update: *Resolved at commit fa36824. Minting more USD against a disabled collateral has been disallowed.*

Medium Severity

M-01 Missing _REWARDS_PERMISSION in `AccountModule`

The `AccountModule` contract contains the `grantPermissions` function, which sets new `permission`s to a specific `user`. The function itself can only be called by either a user with `_ADMIN_PERMISSION` or the `owner`.

Role segregation is meant to let the user have ownership of an account in a cold wallet, while managing operations in the system with hot wallets or trusted accounts. One of the different roles is the `_REWARDS_PERMISSION` which is meant to be held by whoever calls the `claimRewards` function in the `RewardsManagerModule` contract. However, when granting permissions, the `grantPermission` function checks whether the permission that is going to be set is valid by calling `_isPermissionValid`. However, this function is lacking the `_REWARDS_PERMISSION`. For this reason, this permission cannot be set and the only actors able to claim rewards will exclusively be the owner and the `_ADMIN_PERMISSION` granted accounts.

Consider adding the `_REWARDS_PERMISSION` in the `_isPermissionValid` function.

Update: *Resolved in pull request #1379 at commit 45cc883. Tests have been added at commit da5a191 to ensure permissions are always checked against all the ones defined in the future.*

M-02 Lack of feature implementation leads to underflow

Market managers can deposit collateral directly into a `Market` in order to boost liquidity beyond the delegated collateral from the pools that support them. The maximum amount withdrawable is defined as the sum of `creditCapacity` and the total collateral deposited directly into it. This means that a market can withdraw an `amount` larger than its `creditCapacity`, provided there is enough extra collateral deposited.

When withdrawing USD from the Market, the withdrawable `amount` has to be lower than the maximum amount withdrawable described earlier.

However, if a market attempted to withdraw an `amount` larger than its `creditCapacity`, regardless of how much collateral has been deposited, it will revert due to an underflow on the `creditCapacity` update.

The underlying issue seems to be that there is no strategy implemented to decide how to allocate extra debt withdrawn through the market using the boosted credit capacity provided by depositing collateral directly into the market.

Consider modifying the accounting algorithm to properly manage withdrawal amounts that draw from the extra collateral deposited apart from the delegated `creditCapacity` from pools.

Update: *Resolved at commit d1c7488, pull request #1465 at commit 515daea and pull request #1466 at commit d2fb4d7. The `creditCapacityD18` has been changed to `int` and the code was refactored to support such change. Now the `getWithdrawableMarketUsd` will report the sum of the credit capacity and the deposited collateral value if positive, and zero otherwise.*

M-03 Escape hatch mechanism cannot be used for grieved markets

An attacker might create a very large amount of pools, each of them delegating the minimum amount of collateral to a particular market. Since pools are ordered from least to most debt absorbing capacity within a `heap` data structure, it is possible that a market might become grieved (i.e., not able to ever distribute all of its debt to the supporting pools).

There currently is an escape hatch mechanism in place - a `maxIter` parameter that is supposed to be configurable by a market manager in order to mitigate this issue.

However, there is no entry point available for a market manager to call `distributeDebtToPools` with a custom `maxIter` value.

Consider making the necessary adjustments to `MarketManagerModule` in order to make the escape hatch mechanism effective.

Update: *Resolved at commit 08c008c. The `hardhat/console.sol` import has been removed in pull request #1351.*

M-04 Inconsistent debt association

When a market decides to associate debt directly to a specific account, it first proportionally reduces the whole distribution debt by `amount` among all pool vault participants, so that the same `amount` can be assigned directly to the target `accountId`, keeping the total debt within the distribution constant.

If the target `accountId` is not part of the distribution, then the final assigned `debt` for the account will equal the target `amount`. However, if `accountId` is an existing vault participant, a proportional amount of debt will be reduced from their position first, resulting in different total outstanding debt values which depend on the percentage of shares within the distribution owned by `accountId`.

As an example, if there are 10 vault participants each holding 10 snxUSD worth of debt and one of the participants was assigned 10 snxUSD of debt directly, first all participants will see their debt reduced to 9 snxUSD and the target `accountId` will end up with 19 snxUSD. If there were 5 vault participants instead, they would see their debt reduced to 8 snxUSD and the target account will end up with a total debt of 18 snxUSD. The less participants on the distribution before debt is assigned, the more of a discount `accountId` will receive.

If this is the intended design, consider explicitly writing why such a scenario is intended in the docstrings. Otherwise, consider reducing debt among all distribution participants except `accountId`, so that the final debt values are consistent regardless of what percentage of the vault belongs to it.

Update: *Resolved at commit ba87e8d. The Synthetix team added specific documentation to explain why this is the desired behaviour and not an issue.*

M-05 Uninitialized implementations can brick the system

The protocol is meant to have proxies behind contract implementations, so that the implementations can be initialized through delegate calls starting from the proxy itself. However, the implementations on their own are left uninitialized, and anyone can claim ownership of those. An example is the `InitialModuleBundle`.

Moreover, the upgradability mechanism has the upgrade logic in the implementation itself, exposing potentially dangerous upgrade functions to malicious actors that might attack directly from the implementation contracts.

This `InitialModuleBundle` is supposed to be part of:

- The `AccountRouter`
- The `CoreRouter`
- The `USDRouter`

This module provides ownership and upgradability features. The upgradability feature is UUPS, meaning that the upgrade logic is in the implementation itself. Routers are meant to be called through UUPS proxies, which means the proxy storage will be used instead of the routers.

However, implementations (specifically the router implementation) do not disable initialization in the implementation itself, but are left uninitialized.

Whoever calls the `initializeOwnerModule` function directly against the router implementation will take ownership of it. After claiming ownership, the attacker can call `upgradeTo` against the router itself with a malicious implementation that passes the rollback safe test. This will not modify the `implementation` storage value on the proxy, but will change it on the router storage. We were not able to find a way to actually call `selfdestruct` directly while upgrading, since it would be caught on the rollback test.

However, there is an unused `CommsMixin` library which, if imported by any module at any time, would allow the attacker to call it and perform a `delegatecall` against the malicious implementation by reading it from the modified storage, which could trigger a `selfdestruct`, destroying the router implementation and thus bricking the entire system.

It is strongly recommended to include a lock-in mechanism in the implementations contract to prevent initializations after construction, so no user is allowed to take control of them nor use them maliciously. Consider referring to the `_disableInitializers` function of the OpenZeppelin library within the implementations' constructors to achieve an equivalent functionality.

Update: *Resolved in pull request #1423 at commit 08cf342. The Synthetix team decided to remove owner initialization from the implementation and have it instead initialized on the Proxy constructor directly. Moreover, the team removed the `onlyOwnerIfSet` modifier from the `Ownable` contract, to avoid it being bypassed when the owner is set to the zero address. This completely fixes the issue as long as no upgrades or new contracts are allowed to gain ownership directly from the implementation and no arbitrary `delegatecalls` are made to `proxyStore().implementation` in future upgrades.*

Low Severity

L-01 Incorrect handling of collateral tokens with more than 18 decimals

New collateral tokens can only be added by the system owner. In general, ERC-20 implementations do not have more than 18 decimals, but if at any point a new collateral type with more than 18 decimals is added, its entire balance could be drained by a malicious third party due to a precision loss when converting token native amounts into system amounts (18 decimals).

This conversion is performed within the `CollateralConfiguration` storage library. If the collateral had for instance 27 decimals, any amount lower than `1e9` would be returned as zero by that function. For example, if `tokenAmount = 1e8` and the token has 27 decimals:

```
systemAmount = 1e8 * 1e18 / 1e27 = 0
```

An attacker could create an empty account and perform a withdrawal of an amount lower than `1e9`, resulting in a system amount of 0 when converting it to 18 decimals. No amount would be deducted from their collateral since it has been calculated to be 0. However, the attacker would receive the specified amount via parameter since the amount transferred is not the system amount but rather the arbitrary `tokenAmount` specified as a parameter.

If done in a loop, the entire contract balance can be drained.

Consider enforcing that new collateral types have at most 18 decimals in order to be added to the system, refactoring the conversion formula, or recalculating the amount to be transferred from the `systemAmount` in order to account for such a scenario, despite its low likelihood.

Update: *Resolved in pull request #1406 at commit a967821. The new accounting mechanism includes cases in which the `decimals` function call reverts, and in this case the contract assumes `18` as decimal values. Moreover, if a token has more than 18 decimals, the contract now downscales it to `18` only if no precision is lost while doing so.*

L-02 Gas exhaustion prevention is not correctly implemented

When cleaning expired locks from a given account, there is a mechanism to prevent gas exhaustion in case the collateral locks array is too large.

The `if` statement in line 125 is checking whether the `offset` or the `items` variables are greater than `locks.length`. However, it should check whether they are greater than `locks.length - 1` instead.

Moreover, the `items` parameter is only being used to check that its value is capped at the length of the `CollateralLock` array. There is no enforcement to ensure the loop stops after cleaning `items` locks. This causes the loop to always clean all locks from `offset` up to the array length, invalidating the designed gas exhaustion prevention strategy.

Consider limiting the loop iterations to `items` instead of the full array length in order to enforce an effective limit, and fixing the inequality to the correct length size.

Update: *Resolved at commit 99f5288.*

L-03 Expired collateral locks can be created

When creating a collateral lock, there is no check that the `expireTimestamp` value is in the future, allowing the creation of collateral locks that are already expired.

Consider enforcing `expireTimestamp` to be in the future, to ensure the locking makes sense and does not use storage inefficiently.

Update: *Resolved at commit 99f5288.*

L-04 Minting new accounts skips the safety check

When creating a new account, an NFT is minted.

The NFT contract has two ways to transfer the tokens, either by transferring them directly with the `transferFrom` function or with the `safeTransferFrom` function that checks whether the `to` address is an EOA or a contract. If the `to` address is a contract, the `safeTransferFrom` will ensure that it adheres to the `ERC721Receiver` interface.

The `ERC721Receiver` interface is meant to avoid sending tokens to contracts that have no internal logic to eventually handle further transferring of tokens out of the contract itself, or that are unaware that `ERC-721` tokens can be sent to them.

However, the `mint` functionality is missing a corresponding `safeMint` function, where the receiver of a freshly minted token is treated as the recipient of a `safeTransferFrom` call.

Consider adding a `safeMint` function that mimics the `mint` function, but adding a layer of protection by ensuring that if the recipient is a contract, it complies with the `ERC721Receiver` interface.

Update: *Resolved at commit e828d5e. The module now has a `safeMint` feature and it is used on account creation.*

L-05 Incorrect event emission parameter and return value

When liquidating an entire `vault`, the amount that the liquidator receives as a reward equals the entire collateral held within that `vault`. However, when setting this information into the `VaultLiquidation` event parameters, it is incorrectly set to `debtLiquidated`. This value is also used as a return value, which may affect other systems calling this function.

Consider assigning the correct reward amount to the `liquidationData` structure in order to avoid emitting and returning incorrect information.

Update: *Resolved at commit bbb9a52.*

L-06 `lastRewardPerShare` is set on non-existent accounts

The `getClaimableRewards` function of the `RewardsManagerModule` can be used as a getter if called through a static call. However, if called with `call`, the function will trigger an `updateRewards` execution that will change the state of the contract. Specifically, it will update the `pendingSendD18` rewards for the given `accountId` and the `lastRewardPerShareD18` saved into the `RewardsDistributionClaimStatus` object.

However, the existence of `accountId` is never checked and if a non-existent account is used, the `pendingSendD18` parameter will be set to 0 (since the account has no shares in the

account debt distribution) but the `lastRewardPerShareD18` will be set to the current `rewardPerShare` value.

If at some point the user exists, these values will be overridden in the `delegateCollateral` function whenever a user joins a specific pool with a specific collateral. Even if this does not imply a security concern, for correctness, consider checking the existence of the `accountId` when calling the `getClaimableRewards` function.

Update: *Resolved at commit a41de81. New tests were added as well to cover this situation.*

L-07 Wrong or erroneous results in getters

There are instances in the codebase where some getter functions that are not used internally return wrong or erroneous results. Some examples are:

- The `currentVaultCollateralRatio` of the `Pool` storage returns `debt / collateral` as a collateralization ratio instead of `collateral / debt`.
- The `currentAccountCollateralizationRatio` function of the same file might divide by 0, causing the transaction to revert.

Even if these getters are not used internally, they might be in the future, or might be a dependency in other protocols that make use of them. For this reason it is important to show consistent and correct results to avoid potential unexpected reverts.

Consider fixing both getter functions and reviewing any other functions that might be used in the future, either internally or externally.

Update: *Resolved in pull request #1443 at commit 92b493c. Both functions have been fixed and renamed.*

L-08 Unclear arithmetic comparison

The `isCapacityLocked` function of the `Market` storage checks whether the market credit capacity is strictly lower than the reported locked debt from the market contract. It is unclear whether this should be strictly "lower" or "lower or equal" instead, changing the `<` to a `<=`.

Consider either specifying why the inequality is strict or fixing it to be "lower or equal". When doing so, consider the case in which the credit capacity is zero and ensure that this does not have a negative impact on functions that rely on this result.

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

> *The current implementation of the code is correct. A market enters the 'capacity locked' state when the amount of credit capacity available is insufficient to cover the locked amount specified by it. It is technically sufficient for a market to have exactly the amount of credit capacity required by the lock. Furthermore, if the credit capacity is equal to 0, and the market specifies a locked amount of 0, then the capacity shouldn't be locked.*

L-09 Vault liquidations do not reset storage

When performing a full vault liquidation, the vault is reset, which increments the epoch counter internally so that a new fresh storage space is generated.

However, the old `VaultEpoch` storage is left untouched, containing old collateral and debt values. This might cause problems in the future since the whole system is upgradable.

Consider reseting storage values to zero upon full vault liquidations, in order to make sure the state is clean and no problems might arise in future system upgrades.

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

> *While storage slots are not actually cleared, the incrementing of the `currentEpoch` serves the same purpose. We do not anticipate using the old epoch data, though keeping it available could be useful in a future feature. It may also reduce gas usage in relevant functions.*

L-10 Rewards might be claimed but never transferred

When claiming rewards, the actual transfer of funds is handled by the `distributor` when the `payout` function is called.

According to the `IRewardDistributor` interface, which every `distributor` must conform to, the `payout` function returns a `boolean` which we can only assume indicates whether the payout was successful.

If indeed the reward transfer fails but the `distributor` does not revert (returning `false` instead), the rewards will be accounted as successfully claimed, not allowing the user to ever reclaim those rewards.

Consider checking the return value for the `payout` function and reacting accordingly to avoid loss of funds, or improving documentation regarding the `IRewardDistributor` interface in case it is supposed to have a different meaning.

Update: *Resolved at commit 3393d74.*

L-11 Implicit downcast can lead to unexpected reverts

The `V3CoreProxy` interface defines the return value for the `getPreferredPool` function as `uint128`, but the actual implementation returns `uint256`.

If the `poolId` is any value `> type(uint128).max`, the implicit downcast will make the transaction revert, leading to potentially dangerous and unexpected outcomes.

Consider being consistent with the data types returned on interface definitions in order to avoid unexpected reverts.

Update: *Resolved at commit 521cb08. All instances have been changed to `uint128` for consistency.*

L-12 Account existence is not checked

When a `Market` calls the `associatedDebt` function in the `AssociateDebtModule`, it passes several parameters as input, and amongst them is the `accountId` of the user to whom the debt should be assigned.

However, there is no check that the `accountId` actually exists. Fortunately the transaction will just revert in the `_verifyCollateralRatio` function since the `collateralValue` will be 0 and the `debt != 0`.

Consider adding a proper check for account existence instead of relying on the revert of the collateral ratio check.

Update: *Resolved at commit 02b8bb9.*

L-13 Market debts are not independent

When calling `distributeDebtToVaults` within the `Pool` storage library, debt needs to be brought down from supported markets into vaults. Each pool can support multiple markets, and these pools need to be rebalanced if their capacity gets filled.

While iterating over the different market configurations supported by a specific pool, credit capacity is calculated based on their weights, which is coherent with documentation. However, market debt is also calculated based on weights, which does not seem accurate since markets can arbitrarily use their credit capacity independently from one another. If a pool equally supports two markets and one of them has not yet used any credit capacity while the other has maximized it, the average debt will be equal for both, which could potentially affect borrowing capacity for both.

If this is intended behaviour, consider adding docstrings to this function in order to explain the mechanics behind it. Otherwise, consider tracking the exact debt reported by each market that should be passed down to the supporting pool, instead of calculating it based on weights.

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

> *This is the intended behavior. The debt is redistributed among all of the markets in a pool when the pool rebalances. This creates very efficient collateral utilization, and is part of the reason why we give the pool owner capability to set `maxDebtPerShare`, preventing a single market from consuming all the credit capacity of a pool.*

L-14 Users are never removed from pools

One of the parameters of the `Account` storage object is the list of `collaterals` that the user has deposited, which is a mapping of instances of the `Collateral` object. Each collateral object has a list of `pools` ids in which the collateral is delegated.

When delegating collateral to a pool, if the user removes all the delegation, the `poolId` is effectively removed from the `pools` list inside the `Account` storage, but it is then always re-added afterwards.

This leads to the impossibility of removing a specific pool from the `pools` list. This has no effects on the protocol as of right now, but it may produce unexpected outcomes in future upgrades that leverage the `pools` list to run meaningful operations.

Consider ensuring that if a user removes all their delegation from a pool over a specific collateral, the corresponding pool id is then removed from the `Account`'s `pools` list.

Update: *Resolved in pull request #1401 at commit b28c027 and pull request #1431 at commit f72f58c.*

L-15 Lack of input validation

The `initialize` function of the `USDTokenModule` does not check if `tokenName`, `tokenSymbol` and `tokenDecimals` have proper values. The internal `_initialize` function of the `ERC20` contract only checks for `name`, `symbol` and `decimals` to not already have a non-null value, but nothing prevents someone from calling `initialize` with all null values and successfully initializing the contract with those. What is worse is that if this happens, the `initialize` function can be called again.

Moreover, the `initialize` function does not call the `_isInitialized` function, needed to determine whether the contract has already been initialized. The `_isInitialized` function checks for `decimals` to be different from 0, and it can be called within the `initialize` function by using the `onlyIfNotInitialized` modifier, inherited by the `InitializableMixin` contract.

Consider using proper checks to avoid null values from being set and reviewing the codebase to use appropriate initialization modifiers in all the initialization functions.

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

> *Not checking `_isInitialized` is intended behavior, in order to accommodate possible protocol updates where the initialization parameters could change. As of right now, the details of such an operation are undefined. Also, due to changes addressing M-05, its not possible to call initialize outside the proxy as non-owner. As per protocol design, the core system must be the owner of the `USDTokenModule` in order for it to be used by the system, so we do not consider this interface to be a risk.*

L-16 Front-run opportunities

In order to call `createAccount` one must provide a non-used id. This means that anyone can front-run a transaction and create a DoS on the create accounts functionality. This is even more feasible if the protocol is on an L2 where the gas cost of minting NFTs is even smaller. The same happens on pool creation.

Not only is a DoS feasible, but the front-run opportunity can be leveraged in other scenarios, as mentioned in H-02.

Consider documenting such behavior or enforcing an internal mechanism to generate incremental ids instead of using those provided by user input.

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

> *For now we have decided not to implement the incremental ID functionality because we deem front-running to be low risk. L2s generally do not have a mempool which can be frontrun, frontrunning on L1 would be expensive (with ambiguous upside), and flashbots is available to privately create an account without frontrunning risk. We encourage the use of specifying IDs to allow for multicalls to be created as part of the account creation process.*

L-17 Legacy market interactions with out-of-scope system

The `LegacyMarket` contract is the migrator contract to move user positions from the out-of-scope Synthetix v2x system to the in-scope and newer Synthetix v3 system. It works by migrating user debt and collateral from the old system to the new one, while also providing the option to convert `sUSD` to the new `snxUSD` token.

Since Synthetix v2x is out of scope for the current audit, we wanted to highlight some interactions that might benefit from reassessing whether more validations should be added, and if not, specifying why those are not needed.

- The `_calculateDebtValueMigrated` retrieves the `totalDebtShares` from the v2x system. If this happens to be 0, an unhandled division by 0 can occur.
- `burnSynths` can return without actually burning under some circumstances. If this happens, the assumption on the correct accounting would be broken and the execution may have unexpected results.
- During a migration, liquidator rewards are collected, v2x vestings are migrated, and any remaining escrow is sent to the `LegacyMarket`. Whether these three operations include all possibly unclaimed rewards and secondary funds that a user might have is unclear. Consider reviewing whether these three operations account for all user funds and no funds can be stuck in the v2x system after a migration.
- When converting `sUSD` to `snxUSD` the `LegacyMarket` calls the `burnSynths` function in v2x. This will effectively burn `sUSD` and also the corresponding debt shares associated with it, correctly reducing the reported debt before incrementing the net issuance given by the `snxUSD` minting. This means that the conversion can happen only if enough debt shares and collateral have been migrated already by other users as

correctly pointed out in the docstrings. However, anybody can buy `sUSD` on other exchanges and consume debt shares given by other user's migrations. Whether the conversion should be restricted only to users that already migrated their positions or not is not clarified. Consider reviewing the current design and eventually specify the intended behaviour in the docstrings.

For each one of these situations, consider whether it is safe to assume that the execution happened as expected, with no further validation. Otherwise consider adding any protection deemed necessary. Moreover there's no feature to convert `snxUSD` back to `sUSD` in case an `sUSD/snxUSD` pool is created. In that case users might need such a feature to correctly close their positions. Consider specifying whether this feature is going to be implemented or not.

Update: *Resolved at commit a0539a0. What has been fixed here is that the `burnSynths` call can return without actually burning, breaking some assumptions. In this case the team will now revert the execution. The Synthetix team acknowledged the issue of `totalDebtShares` potentially being zero by stating:*

> *The only way for the `totalDebtShares` issue to manifest is if an oracle is broken or if all collateral has been migrated. In the latter case, it is not possible to call migrate with a non-zero debt to migrate.*

The last point has been clarified in the docstrings in the same commit above. About the source of all v2x user funds, the Synthetix team stated:

> *Regarding the clarity of the source of user funds, this can be reviewed by observing the collateral function in Issuer, where the source of user funds can be attributed to 1) the SNX in the user's wallet 2) their SNX in the escrow contract and 3) SNX from liquidations. The liquidation rewards are rolled into escrow, and then the escrow is revoked. Therefore, unless v2x changes in some way, all sources are accounted for.*

Notes & Additional Information

N-01 Unnamed return parameters

There are a few instances within the codebase where returned parameters are not named. An example is the associateDebt function within `AssociateDebtModule`.

Consider adding and using named return parameters to improve explicitness and readability.

Update: *Resolved at commit b47434e.*

N-02 Incorrect or missing documentation and docstrings

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

- The `burnUSD` function documentation mentions a `BURN` role that is not mentioned in the permissions section.
- The event parameters for the PermissionRevoked are copy/pasted from the `PermissionGranted` event above.
- In the ICollateralModule:
  - Within the `Deposited` event, 'if' should be 'id' and amount should be `tokenAmount`.
  - Within the `Withdrawn` event, amount should be `tokenAmount`.
  - The `deposit` function emits a `Deposited` event, instead of `CollateralDeposited`.
  - The `withdraw` function emits a `Withdrawn` event instead of `CollateralWithdrawn`.
- Missing docstrings from `VaultEpoch.assignDebtToAccount`.
- In the `Vault`'s docstrings "the them" should be "them".
- `MarketPoolInfo` docstrings are incorrect; they belong to `DistributionActor` instead.

- `IAssociatedSystem` interface description of `KIND_ERC721` is incorrect since those systems wrap `ERC-721` implementations rather than `ERC-20` implementations.
- The `getCollateralTotals` function of the `Account` storage has a docstring that suggests the function returns the collateral type, but that is not true. It should indicate that the returned values depend on the `collateralType` parameter. Additionally, the function has old code commented out which should be removed.
- The UsdBurned event docstrings regarding parameters belong to the `UsdMinted` event instead.
- Incorrect description for the marketId field.
- Missing docstrings for `AccountMigrated` and `ConvertedUsd` events.
- Line 53 of the `ILegacyMarket` interface contains a typo: "se" should be "set" instead.

In order to improve correctness, clarity and readability, consider fixing the reported examples and reviewing the codebase for more instances.

Update: *Resolved in pull request #1437 at commit cf2af6f.*

N-03 Unused or circular imports

Throughout the codebase, imports on the following lines are unused and could be removed:

- `AuthorizableStorage` of `USDTokenModule`
- `UUPSProxy` and `OwnableStorage` of `AccountModule`
- `IUSDTokenModule` of `UtilsModule.sol`
- `IUSDTokenModule` of `VaultModule.sol`
- `Pool` of `Collateral.sol`
- `IAggregatorV3Interface` of `CollateralConfiguration.sol`
- `SetUtil` of `CollateralLock.sol`
- `ParameterError` of `Distribution.sol`
- `SetUtil` of `Market.sol`
- `SetUtil` of `MarketConfiguration.sol`
- `DistributionActor` of `ScalableMapping.sol`

Moreover, there are instances where there are circular imports. For example, the `Account` storage imports `AccountRBAC`, but `AccountRBAC` imports `Account`.

This is not a comprehensive list of all instances. Consider reviewing the codebase and removing unused imports to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #1351 at commit e17bdbb.*

N-04 Wrong or unused dependencies

Throughout the codebase, there are instances where the dependencies are either imported from the wrong path or are not used within the contract. For example:

- The `InitialModuleBundle` contract uses the `core-modules` path to import `OwnerModule` and `UpgradeModule`, while it should be using the `common` path instead in the parent directory.
- Not all the imports of the `SafeCast` library are effectively used in the `AccountTokenModule` contract.

Consider removing unused dependencies and otherwise correcting the paths for used dependencies.

Update: *Resolved at commit 07a5f14.*

N-05 Inconsistent use of implicit int and uint types

Throughout the codebase, there is an 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 can be seen within:

- CollateralConfigurationModule
- Distribution data structure
- `IPoolConfigurationModule` interface, since the return value for `getPreferredPool` is defined as `uint256` while the implementation is defined as `uint`.

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

Update: *Resolved at commit ebb472e.*

N-06 Inconsistent data type selection

Throughout the codebase, some inconsistencies across data type selection for specific relevant values within the system can be found with no clear objective. For example:

- Pool id is stored as a `uint128` data type, while the return data type from `getPreferredPool` provides a `uint256` instead.
- The amountD18 input in the `assignDebtToAccount` function of the `VaultEpoch` contract is an `int256` which is converted to `int128` in line 94 to be assigned to the `consolidatedDebtAmountsD18` mapping, but this holds `int256` values, so there is no need to down cast.

Consider enforcing a consistent data type selection whenever it does not provide an advantage in terms of struct packing or any other gas optimization.

Update: *Resolved in pull request #1437 at commit cf2af6f. The first item has been resolved elsewhere, but in the `main` branch it is not consistent and converted to `uint128`.*

N-07 Unused code

There are instances within the codebase where either variables or specific code are unused. For example:

- When a user deposits a certain `collateralType` for the first time, there is a `boolean` variable in storage that tracks whether it has ever been deposited before. However, this field is never being read again so it is using an extra storage slot and unnecessary readings on each deposit. Moreover, this `boolean` is never set to false again if the amount of one specific collateral type is zero.
- When the `grantPermission` function is called in the `AccountRBAC` contract, the third `if` statement checks whether the `target` address is not already in the `permissionAddresses` set, in which case it will call `add` in the set to insert the value. However, the same check is performed inside the `add` function, making the first check redundant.
- The `ScalableMapping` is not making use of the `mulDecimal` and `divDecimal` functions of the `DecimalMath` library and it is instead handling the down or up scale manually.

Consider removing each of these situations in order to make code more performant and optimized in terms of gas consumption.

Update: *Partially resolved at commit c6d5102 where the redundant code in the `deposits` function has been removed. The use of the `if` statement in `grantPermission` in the `AccountRBAC` contract was deemed correct since the function should not revert if the value was already set. The `scalableMapping` contract is still not making use of the `mulDecimal` and `divDecimal` functions.*

N-08 Inconsistent use of custom errors

Throughout the codebase, there is a predominant use of custom errors. However, there are some instances where they are not used or they are used inconsistently, such as:

- When updating a specific `rewardId` for a certain `accountId`, the revert error message is hardcoded.
- The `onlyIfAssociated` function of the `AssociatedSystemModule` reverts with an unclear `NotInitialized` error.

Consider using custom errors consistently across the codebase.

Update: *Resolved at commits 4abe9d3 and 97336f9. Regarding the hardcoded errors in the `Market` storage library, the Synthetix team stated:*

> *The use of `require` in Market.sol is for sanity/safety checks, so the need to have custom errors here is excessive. The other inconsistencies have been resolved.*

N-09 Interfaces inconsistencies

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

- In the `USDTokenModule` contract, the `mint` and `burn` functions have `target` as a named parameter, while the `ITokenModule` interface calls it `to`.
- The `IAccountModule` interface returns unnamed parameters for the `getAccountPermissions` function, but the implementation returns named parameters instead. Moreover, the `isAuthorized` function has `target` as a named parameter, while the implementation uses `user` instead.
- In line 20 of the `IAccountModule`, `sender` should be `msg.sender` instead.
- The `getCollateralConfigurations` and `getCollateralConfiguration` of the `ICollateralConfigurationModule` are returning named parameters, while the implementation does not.

- The `IUSDTokenModule` interface misses many function definitions which are present in the implementation.
- The defined return value for the `getApprovedPools` function within the `IPoolConfigurationModule` interface uses `uint256[] calldata` when there is no reason to do so, especially in a function without any input parameters. The `PoolConfigurationModule` implementation defines the return value correctly as `uint[] memory`.
- The `getPoolConfiguration` function features a named return parameter on the interface, while it does not on the implementation.
- In the `IV3CoreProxy` the `registerMarket` function returns an unnamed parameter, but the `MarketManagerModule` contract which the interface refers to returns a named one.
- In the `IMarket` interface, the `name` and `locked` functions are defined as `view` but the implementation defines them as `pure` instead.

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

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

> *We can't enforce this right now because there is no solhint rule for it, but we will create the rule in the future.*

N-10 Variable shadowing

In the `getAccountPermissions` function of the `AccountModule` contract, the `permissions` variable is being used as a return parameter and also as a key value for the `AccountPermissions` struct.

In order to improve readability and avoid using the same name for different context and variables, consider changing the name of the returned variable.

Update: *Resolved at commit 26f1c52.*

N-11 Wrong function visibility

The `deposit` and `withdraw` functions of the `CollateralModule` are defined as `public`, but they are not used by any other contract inside the `core` component. Consider making these functions `external` instead.

Update: *Resolved at commit 131fa7f.*

N-12 Unhandled division by zero can occur

The `ScalableMapping.scale` function does not validate that `totalSharesD18` is not equal to zero. Even if this is unlikely to happen, this would revert the transaction.

This failure would not be handled early, nor would it include an explicit error message. It would unexpectedly stop the execution, reverting without any explicit reason.

Following the "fail early and loudly" principle, consider including specific and informative error-handling structures to avoid unexpected failures.

Update: *Resolved at commit 59a2197.*

N-13 Duplicated code

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

- Within the function `updateCreditCapacity`, the total amount of collateral within the `collateralAmounts` `ScalableMapping` is manually calculated, when such a function is already implemented.
- The custom internal function `_onlyWithPermission` within the `IssueUSDModule` could be removed in favor of using the one defined in the `Account` storage library.
- When a market associates debt directly to an account, collateral value is calculated manually instead of calling the dedicated `currentAccountCollateral` function.
- The `currentVaultCollateralRatio` function is repeating the same code present in the `currentVaultDebt` function.

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

Update: *Resolved at commit 01b7e08. The second point in the list has also been addressed elsewhere and, at the time of writing, is already resolved in the `main` branch.*

N-14 Lack of indexed parameters

Throughout the codebase, some events do not have their parameters indexed. For example, on line 29 of the `IAssociatedSystemsModule` interface (out of scope, but imported into `AssociatedSystemsModule`), the `proxy` parameter should be `indexed` since it is the main entry point for the associated system.

Consider indexing these event parameters to avoid hindering off-chain services searching and filtering for specific events.

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

> *We index parameters based on whether or not the parameter is likely to need to be filtered in a `queryFilter` request. Regarding the example unindexed property given 'proxy', the proxy is the outcome of the operation and would only be useful to index if someone was trying to find all the events indexed to be associated with the proxy address after its created. However, it is more useful to index the other properties, such as 'id', which is important for finding the proxy address from the ID.*

N-15 Gas optimizations are possible

There are several instances throughout the codebase where a refactor might improve gas consumption without sacrificing security. Some examples are:

- When depositing market collateral, authorized access is checked after converting the `tokenAmount` into `systemAmount` (18 decimals). This performs an external call to check the token decimals which can be avoided if authorized access is checked first.
- Line 159 in the `LiquidationModule` is redundant, since `rawVaultDebt` can never be negative at this point because isLiquidatable would have returned false and reverted earlier. It can be safely casted to `uint` without checking whether it is a non-negative value.
- In line 77 of `RewardDistribution`, the contract is first defining the variable `totalSharesD18` but then is not using it in line 89.
- When calculating the total deposited collateral value in USD for a `Market`, it loops through all `DepositedCollateral` entries and queries the current USD price through the Oracle. Consider first checking whether the amount of collateral is larger than zero, in order to avoid querying the Oracle in that case.
- The `accumulateActor` function is called as a ticker from several parts of the system, with the goal of updating a specific actor's last value per share and recovering the value change since the last interaction. In order to update this value in storage, the actor's

number of shares is read from storage and re-updated to the same value in order to leverage the `setActorShares` logic which includes updating this inner value (`lastValuePerShareD27`). Consider isolating this piece of logic in a separate internal function so that it can be reused without the need for two unnecessary SSTORE opcodes (updating actor's shares and total shares within the `Distribution` since those values have not changed).

- In the `ScalableMapping` storage at the `get` function, the `self.totalSharesD18` is first read, but then it is read again internally in the `totalAmount` function called inside.
- In the `createLock` function of the `CollateralModule` contract, the account is first loaded internally in the `onlyWithPermission` function, and then again in the next line.
- In the `RewardDistribution` storage, the `block.timestamp` is first casted to `int256`, then to `int128` to be stored in the `curTime` variable, and then it is converted back again to `uint256` in the `if` statement.
- When calling `distribute` within `RewardDistribution`, the first step is to read total shares from storage and keep the value on the local `totalSharesD18` variable. However, it is then read again from storage instead of using the cached value.
- When migrating a position from v2x within `LegacyMarket`, consider creating the account on V3 before attempting to gather all assets from v2x, including vesting rewards, since the selected `accountId` might already be used, saving the user some gas by reverting early if this is the case.

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 if there are other parts of the codebase that might be affected, and if it would imply changing the security assumptions.

Update: *Resolved in pull request #1445 at commit e957eac. Some changes have been applied as part of other issues and are already present in the `main` branch.*

N-16 Code style and structure inconsistencies

There are instances of incorrect code style or structure that is inconsistent with other parts of the codebase. For example:

- The majority of structs are defined within the `storage` directory, inside each individual `.sol` file. However, some structs such as the `AccountPermissions` or the `LiquidationData` are stored in the interface definition instead.

- The `findMarketWithCapacityLocked` function of the `Pool` storage returns a `lockedMarketId` named parameter, but what the function actually returns is an instance of `Market` storage, and not its id.

Consider fixing and improving the general clarity of the codebase by being consistent with the places in which structs are defined as well as the names given to the returned variables of the functions.

Update: *Partially resolved at commit 7c4823e. Regarding struct definitions, the Synthetix team stated:*

> *The few which are not in the storage directory are only used for external interface purposes. For example, `LiquidationData` is only used to return result information about the liquidation, and never actually touches the contract storage.*

N-17 Reserved slots are misplaced

The `__slotAvailableForFutureUse` variable of the `Account`, `RewardDistribution` and `Vault` storages is meant to be used in the future to host other extra parameters. However, those are placed in the middle of their parent struct definition.

In order to improve the overall quality and readability of the codebase, consider efficiently packing all the struct storages and placing the reserved slot at the end of the struct definition. Notice that if such change is pursued and the structs are reorganized, it is important to avoid any potential storage collisions if the protocol is already deployed on some testnets or other networks.

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

> *These reserved slots exist in the middle of structures because certain values are less than the size of a full slot due to previous upgrades on testnets.*

N-18 Lack of event emission

The following functions do not emit relevant events after executing sensitive actions.

- `setPauseStablecoinConversion`
- `setPauseMigration`

Consider emitting events after sensitive changes take place to facilitate tracking and notify off-chain clients following the contracts' activity. Consider also reviewing the codebase for other occurrences.

Update: *Resolved in pull request #1427 at commit 623a96a and at commit 1746641.*

Conclusions

One critical and five high severity issues were found during this audit. We found the Synthetix V3 system to be very well-designed and robust. Given the overall complexity, we wanted to highlight that there are sensitive dependencies out-of-scope that require special attention during future audits. Moreover, there are several features that are not yet completed, and as such the overall security of the system depends on their correct implementation. Finally, we appreciated that the project came with a very comprehensive test suite and abundant documentation, and the team was very responsive in answering our questions throughout the audit.

Update: *All findings have been either addressed or acknowledged. Some pull requests have been rebased, leading some linked commits to raise a warning about their absence from the main branch. Please refer to the main branch when double checking the presence of such fixes.*

Appendix

Monitoring Recommendations

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

- Account creation failures. This might indicate front-run attacks to prevent `accountId` from being created correctly. This can be achieved by actively monitoring failed calls to the `createAccount` function and also to the migrations in the legacy market. A front-run attack that creates an account before a `migrate` call will lead to a failure in executing the migration.
- Contract calls with null values or with non-existent parameters might signal attempts to initialize storage to meaningful values. An example is the storage initialization of the `lastRewardPershare` of a non-existent `accountId`.
- Transactions that might fail because of an out-of-gas error. If someone creates multiple pools with minimum delegation, the accounting mechanism in the system might run out of gas. This opens the door for DoS attacks.
- Any transaction that reverts might highlight a misuse of the contracts in the system and might hide malicious user behaviour.
- Transactions that deposit ETH or any other used collateral in the system without going through system-defined functions might lead to loss of funds in the system.