Summary

- **Type**
  DeFi
- **Timeline**
  From 2022-07-18 To 2022-08-22
- **Languages**
  Solidity
- **Total Issues**
  40 (29 resolved)
- **Critical Severity Issues**
  0 (0 resolved)
- **High Severity Issues**
  2 (2 resolved)
- **Medium Severity Issues**
  6 (5 resolved)
- **Low Severity Issues**
  12 (11 resolved)
- **Notes & Additional Information**
  20 (11 resolved)

Scope

We audited the “neptune-mutual-blue/protocol” repository at the “73fc82fbe0d1388867b7df669983fe42760daeb1” commit.

In scope were the following contracts:

```
- contracts/core/token/NPM.sol
- contracts/core/Protocol.sol
- contracts/core/cxToken/cxToken.sol
- contracts/core/governance/Witness.sol
- contracts/core/governance/Governance.sol
- contracts/core/governance/resolution/Finalization.sol
- contracts/core/governance/resolution/Unstakable.sol
- contracts/core/governance/resolution/Resolution.sol
- contracts/core/governance/resolution/Resolvable.sol
- contracts/core/governance/Reporter.sol
- contracts/core/ProtoBase.sol
- contracts/core/delegates/VaultDelegateWithFlashLoan.sol
- contracts/core/delegates/VaultDelegate.sol
- contracts/core/delegates/VaultDelegateBase.sol
- contracts/core/liquidity/VaultLiquidity.sol
- contracts/core/liquidity/WithFlashLoan.sol
- contracts/core/liquidity/VaultFactory.sol
- contracts/core/liquidity/Vault.sol
- contracts/core/liquidity/VaultBase.sol
- contracts/core/liquidity/VaultStrategy.sol
- contracts/core/policy/Policy.sol
- contracts/core/claims/Processor.sol
- contracts/core/store/Store.sol
- contracts/core/store/StoreBase.sol
- contracts/libraries/StakingPoolLibV1.sol
- contracts/libraries/CoverUtilV1.sol
- contracts/libraries/StakingPoolCoreLibV1.sol
- contracts/libraries/GovernanceUtilV1.sol
- contracts/libraries/BondPoolLibV1.sol
- contracts/libraries/RoutineInvokerLibV1.sol
- contracts/libraries/StrategyLibV1.sol
- contracts/libraries/PolicyHelperV1.sol
- contracts/libraries/AccessControlLibV1.sol
- contracts/libraries/ValidationLibV1.sol
- contracts/libraries/VaultLibV1.sol
- contracts/pool/Bond/BondPool.sol
- contracts/pool/Bond/BondPoolBase.sol
- contracts/pool/Staking/StakingPoolInfo.sol
- contracts/pool/Staking/StakingPools.sol
- contracts/pool/Staking/StakingPoolBase.sol
- contracts/pool/Staking/StakingPoolReward.sol
- oracle/contracts/NpmPriceOracle.sol
```

Not all components of the system were in-scope. Most notably, the cover lifecycle and strategies were not in-scope but interact closely with many in-scope functions. We assumed the out-of-scope contracts work as documented and mainly focused on in-scope contracts. Despite the restricted scope, we did find certain issues in out-of-scope files and included them in the report accordingly.

System overview

The main purpose of the Neptune Mutual Protocol is to provide parametric insurance products with focus on covering smart contract hacks.

At a high level, the core system comprises the protocol, storage, NPM token, vaults, policies, cover tokens, incident reporting and claim processor. In addition there are bonding, staking, reassurance, price oracle and strategies. Below, we describe each component and some key system-level features.

Components

Protocol

The `Protocol` contract is used for role-based access control and exposes functions to upgrade the system.

Storage

The system uses the Eternal Storage pattern for upgradeability so almost all variables are stored in the `Store` contract except for `reentrancyGuard`, `Ownable`, and a few others.

NPM token

NPM token is a pausable ERC20 token with a total supply cap and the owner has the ability to mint tokens under that cap.

Vaults

When a new cover is created, a fresh vault will be deployed by the `VaultFactory`. Vault contracts store the stablecoin funds associated with the cover. Vaults are not upgradeable, thus most core logic is contained in vault delegate, which is upgradeable.

Insurance underwriters can add liquidity to vaults in exchange for Proof-of-deposit (POD) tokens, while having NPM staked at the same time. After an incident, claims will be transferred out from the vault to the claim processor. In order to create additional revenue for liquidity providers, vaults also (1) act as an ERC3156 flash loan lender and (2) execute investment strategies that provide a portion of available liquidity to protocols like Aave and Compound. Separate strategy-specific contracts can move liquidity out from vaults to lending protocols.

There is a limited time window where liquidity can be withdrawn. Additionally, any deposit prevents subsequent withdrawals for a few blocks. However, liquidity providers can still exit their position by selling the POD tokens, which are freely transferable.

Policies

One can purchase a cover `Policy` via the Policy contract. Based on the chosen amount and duration, a fee will be computed and required to be deposited in exchange for the corresponding amount of a cover token. The particular cover token depends on the expiry month of the policy. It's worth noting that you cannot purchase a cover for the future – it will start immediately after a pre-configured lag. The `PolicyAdmin` configures policies.

Cover tokens

Cover tokens are non-transferrable ERC20 tokens represented by the `cxToken` contract. `cxToken` is redeemable for an equivalent amount of stablecoin payout in the case of a resolved incident before its expiry date at the end of a month. There is a lag period from the date of purchase to the effective cover start date in order to prevent attackers buying covers right before/after a hack and emptying the liquidity. This means that freshly bought tokens are not redeemable immediately.

Incident reporting

Anyone can report an incident by staking a minimum required amount of NPM tokens. This changes the normal product status which stops any policy purchase and withdraws liquidity from strategies. During the reporting period, anyone can refute a claimed incident by staking an equivalent amount of required NPM tokens. This results in two competing camps of votes where any witness can add their vote to either camp via staking any amount of NPM tokens. A natural resolution will come about at the end of the reporting period with the majority votes. However, the protocol reserves the right to overwrite any voting result by an emergency resolution.

To incentivize fast submissions the system pays an additional reward to the first reporter of the winning camp. Importantly, the incident date for the purposes of policy coverage is the time of the first report, not when the alleged incident occurred. This means that some policies may expire after the real incident but before the first report, which is why timely reporting is crucial.

> *We would like to highlight that the attacker can be the reporter if they are willing to. In principle, nobody can report faster than the attacker as the attacker knows when the incident happens in advance. This does not break the mechanics. This gives the attacker a non-stolen reward in addition to stolen funds, although arguably, the reward may be considered justified payment for reporting the attack.*

Once the incident is resolved, the losing camp loses their entire stake. Part of this stake is burned and the remaining part is distributed among the winning camp. There is a deadline for winning stakers to claim their rewards (their share of the losing camp's stake). After this deadline, they will still be able to retrieve their original stake but the corresponding rewards will remain in the system until a recovery agent withdraws them.

If the incident was deemed to have been correctly reported, funds are transferred to the claim processor contract for distribution to policy holders.

Incidents can only be reported and resolved one at a time. This has some interesting implications. In particular, if an invalid report is raised, it would prevent valid reports until it is resolved. Moreover, voters are expected to independently ascertain whether an incident has occurred. This may create additional confusion where some votes occur before an incident, while others occur afterward. The privileged roles in the system are expected to use their powers to minimize and recover from such inconsistencies when they arise.

Claim processor

The claim processor receives funds from a vault and handles payouts during a pre-configured claim period. During each claim, a proportion of the payout is transferred to the treasury as a platform fee, out of which, a commission is rewarded to the first reporter. The protocol can deny claims to any suspicious account during the reporting period of a particular incident via blacklisting.

Price oracle

The price oracle derives a TWAP-based NPM token price from a UniswapV2 NPM-Stablecoin pool. It calculates the NPM-Stablecoin LP token price by the formula described in Fair Uniswap's LP Token Pricing. This is used solely to compute the bonding price of the LP token in terms of NPM tokens.

Bonding

A bonding pool is used to bootstrap protocol-owned liquidity. Users sell their UniswapV2 NPM-Stablecoin LP tokens to the protocol in exchange for discounted and locked NPM tokens. The LP tokens are transferred to an address controlled by the protocol owners who can hold or redeem the tokens. Obtained stablecoin liquidity can be used to support cover pools and the NPM liquidity may recirculate back to the bonding pool however this is not enforced by the code.

NPM and POD staking

NPM and POD staking pools are created to encourage participants to obtain and hold those tokens. Any project can pre-fund a staking pool with its own ERC20 reward tokens that will be released to stakers at a fixed rate until the rewards are depleted.

Reassurance

The reassurance fund is provided by cover owners to strengthen the confidence in the cover's safety. If an incident occurs, the funds will be used to mitigate the losses of liquidity providers. It also reduces the premium paid by cover buyers.

System-level features

There are several system-level features

Pausability

Each contract is meant to be pausable. Pausing occurs in two places: (1) the storage and (2) the `Protocol` contract. It should be noted that while paused, the protocol will stop updating, but any time-based logic is unaffected. This means that, in addition to freezing funds, it could influence the outcome of incident reports, insurance claims, or reward assignments.

Recoverability

The recovery agent can transfer ETH or any ERC20 token from any contract, including `Vault` and `StakingPool` that hold substantial funds.

This design can mitigate many potential vulnerabilities. In particular, any funds that cannot be withdrawn immediately can be managed by the recovery agent.

Privileged Roles

The system has a large number of very privileged roles, as described in the Security document. Users must trust the holders to exercise their powers wisely and fairly and to protect their corresponding cryptographic keys.

Administrator

This is the `NS_ROLES_ADMIN` role, which grants all other roles. This gives it complete control over the system.

Cover manager

This is the `NS_ROLES_COVER_MANAGER` role, which configures various cover parameters, including the claim period, the reporting stake requirements, the reporting commission, the policy fee rates, etc. It also includes the ability to blacklist particular addresses from making claims.

Cover owner

This role can add new products to an existing cover and reassurance funds to covers they own. It can also update the user whitelist for a cover. However, the cover owner cannot decide whether the whitelist is enabled or not.

Governance admin

This is the `NS_ROLES_GOVERNANCE_ADMIN` role, which can overrule any resolution within the cooldown period. It can also configure the cooldown period and disable cover purchases.

Governance agent

This is the `NS_ROLES_GOVERNANCE_AGENT` role, which resolves and finalizes reports to advance the reporting process through its phases. Note that this means that incident resolutions can be stalled by an inactive governance agent. It can also update the list of cover creators.

Liquidity manager

This is the `NS_ROLES_LIQUIDITY_MANAGER` role, which can update the list of strategies and configure various parameters related to strategies. It can also transfer reassurance funds to vaults when needed to payout claims.

It's worth noting that adding a strategy does not make it immediately functional. The strategy must also be granted the Protocol Member role by an Upgrade Agent or Administrator.

NPM token owner

This role can mint new NPM tokens to any address within the total supply cap. It can also pause the NPM token and recover mistakenly sent funds from it.

`Store` owner

This role can pause and unpause the `Store` contract and recover mistakenly sent funds from it.

Pause agent

This is the `NS_ROLES_PAUSE_AGENT` role, which can pause the `Protocol` contract.

Protocol member

Every contract in the system must have this pseudo-role in order to have write access to storage. With this role they can write to the `Store` contract directly, giving them complete control over the system. The protocol members are set by the upgrade agent.

Recovery agent

This is the `NS_ROLES_RECOVERY_AGENT` role, which can transfer all ETH and ERC20 tokens from any contract, including contracts that store user funds like `Vault` and `StakingPool`.

Unpause agent

This is the `NS_ROLES_UNPAUSE_AGENT` role, which can unpause the `Protocol` contract.

Upgrade agent

This is the `NS_ROLES_UPGRADE_AGENT` role, which can add or remove contracts from the protocol, effectively granting or revoking the Protocol member role, and replace the logic contracts with new ones. Thus, it has complete control over the system.

Cover creator

This role can create new covers and products.

Findings

Here we present our findings.

High Severity

Conflated staking pool reward balances

Each staking pool specifies its own reward token and corresponding balance in the same aggregate contract. When retrieving this value, the token balance of the aggregate contract is returned. Since there could be multiple staking pools with the same reward token, this could include balances from other pools. It could also include any reward token balances that were directly sent to the contract.

Moreover, current user rewards could also be overstated, which would prevent users from claiming the last rewards. Since rewards are claimed when withdrawing stake, anyone could prevent users from unstaking by directly sending reward tokens to the staking pool contract. Any non-zero amount would be sufficient to trigger this scenario. If this occurs, a recovery agent could still retrieve the funds from the aggregate pool contract and distribute them as desired, although it is not clear how they should distribute the remaining rewards.

Consider reading the pool balance from the saved record.

Update: Fixed as of commit `8b660b13cf9fbcde0bfedb3819dbb670ba74b09a` *in pull request #156.*

Risk of insufficient liquidity

When purchasing a cover, the protocol ensures it has enough funds to pay out all potential claimants. The computation of the existing commitments includes all covers expiring in the next 3 months, since this is the maximum policy duration. However, some covers may expire in the fourth month and these would be excluded from the calculation. Therefore, the protocol could sell more insurance than it can support, and some valid claimants may be unable to retrieve their payment.

Consider including the extra month in the commitment computation.

Update: Fixed as of commit `63fce22c67f72cf090ffa124784a3d92935e2d66` in pull request #136.

Medium Severity

Unenforced staking requirement

Adding liquidity requires a liquidity provider to have at least a minimum amount of NPM tokens staked in the vault.

However, the purpose and usefulness of this requirement is unclear, since it can be bypassed. In particular:

- there is no relationship between the amount of PODs created and the size of the stake
- PODs are transferable to unstaked users, so users can provide liquidity without staking
- staked users can exit their entire staked `amount` without redeeming any PODs by calling `removeLiquidity` with parameters `podsToRedeem = 0`, `npmStakeToRemove = amount`, and `exit = 1` the `exit = 1` is crucial as it allows execution of line 234 of `VaultLibV1.sol`

Consider documenting and enforcing the intended relationship between NPM staking and liquidity provision.

Update: Acknowledged, not fixed. The Neptune team stated:

> *Although we plan to redo the staking requirement logic from scratch, we wish to consider this risk as acceptable for the time being.*

Potential token transfer from unrelated account

The `CoverReassurance` contract contains a mechanism to retrieve funds from an arbitrary account, as long as the account has provided a non-zero allowance. This would occur whenever a cover owner can front-run another cover owner's reassurance transaction, allowing them to redirect the funds to their own cover.

Even without front-running, there are multiple reasons an account may have a non-zero allowance, including:

- Their `addReassurance` transaction failed and they didn't revoke the allowance.
- They made an unlimited approval.
- They approved a higher allowance than the amount they eventually transferred.

In all cases, an attacker can retrieve those funds and direct them towards a cover.

A recovery agent could still retrieve the funds from the `CoverReassurance` contract and distribute them as desired, although it is unclear how they would distinguish a front-running attack from one where a cover owner legitimately transfers funds from a different account.

Consider retrieving the tokens from the message sender rather than an arbitrary `account` parameter.

Update: Fixed as of commit `ca55b69c5cdd80bcccdc83dd5d569933f450fa6a` *in pull request #139.*

Incorrect policy fee

There are two discrepancies when calculating a policy fee rate:

- It is always strictly higher than the configured floor.
- The amount of days charged does not account for a non-standard coverage lag period.

Consider updating the calculation accordingly.

Update: Fixed as of commit `84a6fc3167adfb61b6f16666f0ba422b60bc0b2c` *in pull request #159 and commit* `4b929c274100a981107e35d40fbf5b57fabc9be4` *in pull request #196. The Neptune team have chosen not to address the first bullet.*

Parallel access control

The `Protocol` contract inherits the OpenZeppelin `AccessControl` contract, and uses it to define the role hierarchy. It also provides a mechanism for the administrator to grant an existing role to a new address. However, this mechanism functions in parallel to the inherited mechanism for granting roles. This leads to two inconsistencies:

- A role administrator can bypass the `whenNotPaused` restriction by using the inherited mechanism.
- The `NS_ROLES_ADMIN` can use the new mechanism to grant the `NS_ROLES_GOVERNANCE_AGENT`, even though they do not directly administer that role.

Consider ensuring consistency between the two mechanisms. Depending on the desired outcome, this could involve relying on the original mechanism, changing the role relationships, or overriding the inherited `grantRole` function.

Update: Fixed as of commit `1d54d66493e3109c12d610f0231529cbd65b5ba9` *in pull request #157 and commit* `4b929c274100a981107e35d40fbf5b57fabc9be4` *in pull request #196.*

Unable to unstake after finalization

Reporters on the winning camp can unstake their tokens even after the incident has been finalized, albeit with no reward. However, the resolution deadline is not specific to a particular incident and is reset to 0 during finalization. Since the deadline is checked during unstaking, the operation will fail. This means that some successful NPM stakers will be unable to retrieve their funds.

In this scenario, a recovery agent could still retrieve the funds from the `Resolution` contract and distribute them as desired.

Consider recording the resolution deadline with the incident date so it does not need to be cleared during finalization.

Update: Fixed as of commit `6cb6b6064eca18cccee8114cbcefd2455c286ce9` *in pull request #132 and commit* `4b929c274100a981107e35d40fbf5b57fabc9be4` *in pull request #196.*

Unexpected deployer privileges

The deployer address of the `Store` contract is recorded as a protocol member, which allows it to update the storage arbitrarily. The same address is set as the contract's owner role, which allows it to pause and unpause storage updates. We believe these are intended to be the same role, but they are not programmatically connected. In particular, if the `owner` address is renounced or transferred, the deployer will still be able to update storage.

Moreover, it is unclear why the `Store` owner or deployer requires the ability to modify storage arbitrarily.

Consider documenting the role in the Security overview if the role is required. Otherwise, consider renouncing protocol member privileges from the deployer address after the deployment is finished.

Update: Fixed as of commit `0b278019c01dbce22923d0bb6968ddb48bcc3e2d` *in pull request #123. The deployer address is removed as a protocol member, assuming the deployer is the address that calls the* `initialize` *function.*

Low Severity

Able to close non-empty staking pool

A staking pool can be closed without checking if there is any remaining liquidity of either the staking token or the reward token. Once the pool is closed, neither `deposit` nor `withdraw` functions are allowed. Hence, users won't be able to access their funds. However a recovery agent is still able to retrieve both staking and reward tokens and distribute them as desired.

Consider checking for remaining liquidity before closing a pool.

Update: Fixed as of commit `86b0caa0995ffcdbb1deecf8547c9a3db8c23821` *in pull request #160.*

Collision between constants

The `NS_POOL_MAX_STAKE` and `NS_POOL_REWARD_TOKEN` constants are defined to be the same string, which introduces the possibility of unexpected storage collisions. In the current code base they are used with non-overlapping data types, which are saved in different mappings. Nevertheless, in the interest of predictability, consider redefining the `NS_POOL_MAX_STAKE` constant to a unique string.

Update: Fixed as of commit `90f03dce0d24af3affc50d19ac81bbc12b524a4f` *in pull request #161.*

Implicit timing assumptions

To account for the coverage delay, some valid cxTokens may be excluded from making claims. Any coverage that will become active within 14 days but before the incident resolution will be disregarded. This implicitly assumes that no valid cover starts after either of these deadlines (otherwise it should also be excluded). Since the coverage delay and resolution window are configurable parameters, the assumptions may not hold. Consider calculating exclusions based on the specific parameters that are relevant to the incident being processed.

Update: Fixed as of commit `e00b4248768c196a2b5047dcc21d91a2503452ab` *in pull request #162 and commit* `4b929c274100a981107e35d40fbf5b57fabc9be4` *in pull request #196.*

Imprecise bounds

There are several examples where the time windows or value ranges are defined inconsistently. In particular:

- The `getWithdrawalInfoInternal` function of the `RoutineInvokerLibV1` library considers the `end` timestamp to be part of the withdrawal period but the `mustBeDuringWithdrawalPeriod` validation function does not.
- The `StakingPoolLibV1` library prevents withdrawals on the block height where withdrawals can start.
- Neither the `mustBeBeforeResolutionDeadline` function nor the `mustBeAfterResolutionDeadline` function will succeed on the resolution deadline.
- The flash loan fee calculation requires the loan to be strictly less than the available balance, even though the contract claims to loan out the whole balance.

Update: Fixed as of commit `3412b68b9d729d0bc5c3b5860ace7a38a06b9835` *in pull request #167.*

Incorrect NPM threshold

Some operations require an NPM stake that must not exceed a threshold, currently set to 10 billion. However, the total NPM supply cannot exceed 1 billion, making the threshold non-functional. The Neptune team indicated the threshold should only be 10 million. Consider updating the constant accordingly.

Update: Fixed as of commit `78fafa7314793a3b6b5fe40e1c9129c8f8c4f813` *13 in pull request #164.*

Lack of input validation

- The `mustNotExceedNpmThreshold` function should validate `npmStakeToAdd` instead of `amount`.
- The `setPolicyRatesByKey` function in the `PolicyAdmin` contract does not check that `ceiling` is greater than floor, while a similar function `setPolicyRates` does.
- The `initialize` function in the `Protocol` contract does not check the length of the input `values` array.
- When computing unstaking rewards after an incident resolution, the sum of the `toBurn` and `toReporter` rates are not validated to be bounded above by `ProtoUtilV1.MULTIPLIER`.

Consider including the corresponding validations.

Update: Fixed as of commit `5ce4b8d3ff0b0a7eb4f0265b4201c93c43af4f30` *in pull request #172 and commit* `4b929c274100a981107e35d40fbf5b57fabc9be4` *in pull request #196.*

Missing event parameter

The `PoolUpdated` event does not include the `stakingTarget` parameter. Consider including it.

Update: Fixed as of commit `89d30f63d6c43dd3787cd291e31c03a2b712a0a2` in pull request #163.

No unstaking window

After an incident is resolved, successful stakers can retrieve their rewards provided the incident has not been finalized. When the incident occurred, they will have at least the claim period. However, if the incident was successfully disputed, there is no claim period and the incident can be finalized immediately before stakers have been provided sufficient time to claim their rewards. Consider including an unstaking window for this scenario.

Update: Acknowledged, not fixed. The Neptune team stated:

> *For incidents resolved as `false reporting`, we intend to restore the cover status to operational as soon as possible. This flexibility allows us to accomplish a speedier finalization while still allowing the tokenholder community sufficient time to unstake their claim (with reward) on a case-by-case basis.*

Protocol administrator needs to handle external tokens

The protocol administrator is one of the most critical roles with immense privilege in the operation of the entire protocol. For example, only the administrator can re-initialize the protocol, grant key access control roles, as well as set up all staking and bonding pools.

However, when setting up a staking pool, a non-zero amount of reward tokens are required to be pre-transferred to the administrator account and pulled to the contract. This implies that the administrator needs to receive and approve the transaction a priori. This increases the attack surface and may not fit the intended security assumptions for a critical role.

Consider either using a less critical role to perform staking pool initialization or allowing pool initialization without any token transfer.

Update: Fixed as of commit `71fd05996061b9c438c557c92cd888f4f4c9c542` in pull request #173. The Liquidity Manager must now initialise and manage the staking pools. They must also set up the Bond pools.

The `info` parameter might lose information about an IPFS hash

The `info` parameter of the `report`, `dispute`, and other functions assume that the length of the IPFS hash is 32 bytes or shorter. However, that is not the case for CIDv1 where the hash can be longer than 32 bytes and also contain prefixes.

This leads to a data availability issue when NPM holders might be unable to retrieve the incident information from the smart contracts. Consequently, they are unable to decide whether to attest or refute the incident.

Consider using a different data structure for storing an IPFS hash.

Update: Fixed as of commit `5ebb130fe274f0237e368ceaac25751936c1b321` in pull request #165.

Incorrect individual liquidity share

The calculation of an individual's share of liquidity for a particular cover incorrectly uses `values[5]` instead of `values[4]` as the number of PODs. Since this is always zero, the returned share of liquidity will always be zero.

This has no implications within the current code base but would mislead external users that rely on it. Consider using the correct number of PODs in the calculation.

Update: Fixed as of commit `2192646ab5efa95a90521b986c81c05ed04fcd37` in pull request #166.

Variable outside store

In contrast to most of the code base, the last policy identifier is saved directly in the `Policy` contract. However, to maintain continuity and prevent conflicts, any new version will need to import the old value.

Consider saving it in the `Store` contract.

Update: Fixed as of commit `1826fa97f1b325d40b0b3446b384dac35074540f` in pull request #168.

Notes & Additional Information

`transfer` and `send` calls are no longer considered best practice

When `transfer` or `send` calls are used to transfer Ether to an address, they forward only a limited amount of gas. This precludes destination addresses with complex fallback functions. Additionally, given that gas prices for EVM operations are sometimes repriced, code execution on the receiving end of these calls cannot be guaranteed in perpetuity.

There are multiple occurrences throughout the code base where `transfer` or `send` is used to transfer Ether. For instance:

- On line 41 of `StoreBase.sol` Ether is transferred via `transfer`.
- On line 19 of `WithRecovery.sol` Ether is transferred via `transfer`.
- On line 23 of `BaseLibV1.sol` Ether is transferred via `transfer`.

Rather than using `transfer` or `send`, consider using `address.call{value: amount}("")` or the `sendValue` function of the OpenZeppelin `Address` library to transfer Ether.

Update: Fixed as of commit `adf8883628f94a27ae61376e98d112f998029e16` in pull request #187.

Anyone can temporarily DoS a fresh vault

Vaults are deployed by whitelisted cover creators with the `addCover` function. To prevent someone from unbalancing the POD-to-stablecoin ratio immediately after deployment, the Vault detects unmatched stablecoins and reverts on any attempt to add liquidity which effectively disables the vault.

A recovery agent could retrieve the excess funds to re-enable the contract. Nevertheless, to avoid this scenario, consider adding some liquidity in the same transaction as the deployment. Alternatively, consider tracking the stablecoin balance in a variable to mitigate issues caused by direct transfers.

Commit-Reveal voting

The Governance mechanism allows NPM token holders to vote on whether they believe a reported incident is valid. Typically, the rationale for using voting as an oracle is that token holders, who are incentivized to vote with the majority, will treat the truth as a natural Schelling Point. However, since token holders can review the running total, they may instead simply vote with the majority.

This is commonly mitigated with a commit-reveal voting scheme. However, it is also mitigated by the possibility of a governance administrator overruling the vote. We are just noting the practice for your consideration. If the commit-reveal scheme is adopted, votes that are not revealed should be considered incorrect so that users cannot selectively abstain based on the running total.

Copied in dependencies

Dependencies in the lib directory, including `openzeppelin-solidity`, are copied in without any reference in `.gitmodules`. This makes it hard to keep track of the latest versions and easy to accidentally change the code inside.

Consider using `forge install OpenZeppelin/openzeppelin-contracts` for the latest version of the OpenZeppelin contracts.

Update: Fixed as of commit `80f024fb21389e5d29eff9e79a6d0248c6f61183` in pull request #188.

Docstrings not following NatSpec

Across the code base there are several examples of contracts not consistently following the Ethereum Natural Specification Format (NatSpec). Consider following this specification on everything that is part of the contracts' public API.

Some examples include:

- Missing NatSpec for `productKey` at line 151 of `PolicyHelperV1`.
- Discrepancy between `addCover`'s NatSpec in the `Cover` contract and the `ICover` interface the interfaces misses NatSpec for 8th and 9th parameters. Consider using `@inheritdoc` NatSpec tag.
- Return variables are documented using `@param` instead of `@return` Vault.getInfo.

Duplicate modifier

The `unpause` function of the `ProtoBase` contract has two `whenPaused` modifiers. Consider removing the first one.

Update: Fixed as of commit `290c68fd25a1f29673249483982276 84ec834597` in pull request #183.

Duplicate token supply tracking

The NPM token tracks the number of tokens that have been issued. This should be identical to the total supply if the tokens are never burned. It's worth noting that the code base transfers funds to a burner address instead of reducing the supply.

Consider disabling the `burn` functionality so that the total issued amount does not need to be tracked and updated separately.

Update: Not an issue. The Neptune team stated:

> *Given that the NPM token and protocol will be deployed on different blockchains, this is the proper approach. Ethereum is the only chain where `token burn` occurs. The burned tokens are transferred to a specified address on each chain and then bridged back to Ethereum on a regular basis.*

Excessive indirection and coupling

We found this audit to be significantly complicated by data storage reference patterns that hinder the ability to reason locally about each function's behavior in isolation. Although we typically focus on explicit vulnerabilities or specific recommendations, we believe it may be helpful to highlight some general patterns and possible alternatives for your consideration. Naturally, any significant refactor should be thoroughly evaluated and tested.

Key construction

The `StoreKeyUtil` contract has a different function for several supported combinations of data type, operation, number of keys and type of keys. This adds a large amount of boilerplate code. Moreover, not all valid combinations are included, leading to situations where storage keys are sometimes calculated directly and sometimes implicitly specified.

We believe the extra layer of indirection is both complicated and unnecessary, and would be cleaner if storage lookups accepted a generic `bytes` value (that could be hashed to a `bytes32`). This could also remove the need for use-case specific multi-dimensional mapping types.

Meaningful names

Some keys are reused for different variables and are distinguished only by the type. For example, `NS_COVER_PRODUCT` represents whether the product is supported, the product reference, an entry in the cover's product array and the active status of the product. It would be clearer to use different constants for different variables or include a human-readable identifier to distinguish them.

Conversely, contextual values can be reused to highlight commonalities between variables. For example, there are global, cover-specific, product-specific, incident-specific and account-specific variables. These could each be represented by a `context` variable that both identifies the specificity of the key and can be reused between variables with the same context.

Constant-specific functions

There are several examples of duplicated functions that differ only by the relevant constant. It would be simpler and cleaner to pass the constant to a generic function. For example, the access control functions could be replaced by a single function that accepts a role constant. Similarly, generic getter functions can be combined so meaningful helper functions can be distinguished.

Incomplete deletion

When finalizing an incident, an unused record is deleted. Additionally, the first disputer is not deleted. Consider updating the deletions accordingly.

Update: Fixed as of commit `113a6b7ff7fff5730cefedd4d35c7c6cd9f65bbf` *in pull request #186.*

Incorrect array size

The `getCoverPoolSummaryInternal` function creates an array of size 8 but only uses 7 positions. Similarly, the `getInfoInternal` function creates an array of size 11 but only uses 8 positions.

Consider resizing them accordingly.

Update: Fixed as of commit `1826fa97f1b325d40b0b3446b384dac35074540f` *in pull request #168 and commit* `c6b1bb74a299f7c4d6d03484da91e5780fab3faa` *in pull request #190. The arrays have been replaced with structs.*

Misleading comments

Some comments are misleading, and the implementation does not follow the stated intention. For example,

- In line 76, it was stated not to reset the first reporter by incident date. However, the first reporter is not saved by incident date, and it is deleted in line 90 of `Finalization.sol`. Similarly, the commented out lines do not contain the `productKey` and don't correspond to any saved value.
- In line 128 of `Protocol.sol`, it is said that the protocol needs to be paused when the `addMember` function is invoked but in line 136, it ensures the protocol must not be paused.
- The comments describing the `callerMustBeX` functions reference the "sender" rather than the `caller` parameter.
- When initializing the protocol, the burner address must be non-zero but the comment says it isn't necessarily zero.

Consider updating the comments to be aligned with the code implementation.

Update: Fixed as of commit `7d1315614a799ff200f77001aaaaaf91e8ad499a` *in pull request #189 and* `9a3cf2ad7fba096dd5c3cada68b83bf693080baf` *in pull request #196.*

Naming issues hinder code understanding and readability

To favor explicitness and readability, several parts of the contracts may benefit from better naming. Our suggestions are:

- Rename `disablePolicy` to `updateDisablePolicyStatus`.
- Use "timestamp" instead of "date" where relevant throughout the code base.
- Rename "resolution timestamp" to "reporting deadline", to distinguish it from the "resolution deadline".
- Rename `incidentHappened` to `isClaimable`.
- Rename `delgate` to `delegate`.

Product status needs not be incident specific

The status of a cover or product is often queried against its `coverKey` / `productKey` combination. For example, the status needs to be normal before any liquidity event such as adding liquidity, lending out flashloan, reporting an incident , or purchasing a cover proceeds.

However, the product status hashkey depends on an `incidentDate`. When looking up the product status, one first routes to the active `incidentDate` , and then computes the right key to read the internal status. In particular, in the case of a normal status, the active `incidentDate` is always `0` and further checks are no longer necessary. This implicit mechanism is also relied upon for newly deployed products to have a normal status.

Consider refactoring variables such as `ProductStatus` that do not need to depend on the `incidentDate` for its hashkey for clarity and simplicity.

Repeated pause validation

Several functions in the `Protocol` contract have a `whenNotPaused` modifier and `mustNotBePaused` requirement. However, both of these check the pause status of the `Protocol` contract, so one of them is redundant. Consider removing one of them.

Update: Fixed as of commit `7a84b7a6a750224c9f29b2bdbbea84a65c9fdde3` in pull request #170.

Risks associated with the price oracle

An on-chain Time Weighted Average Price (TWAP) oracle is used to derive the NPM and LP token market prices from a UniswapV2 NPM-stablecoin pool. These prices are then used to calculate the amount of NPM tokens returned when users deposit their LP tokens to the bonding pool.

In general, a TWAP price is known to be rather resistant to single block manipulation. However, it is still subject to the risk of multi-block MEV, of which the switch to Proof of Stake may change its feasibility profile drastically.

As the NPM tokens are locked after bond purchase, there is time for the recovery agent to sweep the fund in case of an oracle price manipulation. We recommend close monitoring of the liquidity depth of the NPM-stablecoin pool to mitigate any oracle risk.

Typos

Throughout the code base, there are some incidences of typographical errors in the comments. For example,

- `highy` should be `highly`
- `responsbility` should be `responsibility`
- `Retuns` should be `Returns`
- `indicent` should be `incident`
- extra `as`

Update: Fixed as of commit `4b929c274100a981107e35d40fbf5b57fabc9be4` in pull request #196.

Unexplained dead code

The `CoverUtilV1` contract contains a function that has been commented out without explanation. Consider removing it from the code base or explaining why it is there.

Unnecessary complex code

- The `s.getStablecoin() == address(token) == false` expression on line 245 of `StrategyLibV1.sol` can be replaced with `s.getStablecoin() != address(token)`
.
- Line 272 of `CoverLibV1.sol` casts the variable of type `address` to type `address` The casting can be avoided.
- `isProtocolMember` is defined both in line 257 of `ProtoUtilV1.sol` and line 85 of `StoreBase.sol`.

Unnecessary coupling

When recording an attestation, the stake is recorded against the `who` address, but the reporter is set to the message sender. In both invocations, the `who` parameter is set to the message sender anyway. Nevertheless, in the interest of local reasoning, consider using the `who` address consistently. A similar observation applies to the first disputer.

Similarly, when calculating the future commitment, the number of months to check should depend on the global limit.

Update: Fixed as of commit `63fce22c67f72cf090ffa124784a3d92935e2d66` in pull request #136 and commit `f47e959a2f89e29390164257e7dce298442cff11` in pull request #184.

Unused imports

Throughout the code base many imports are unused and could be removed. Some examples are:

- Line 5 of `ProtoBase.sol`
- Line 4 of `cxTokenFactory.sol`
- Line 7 of `Resolvable.sol`
- Line 5 of `Unstakable.sol`

Consider removing unused imports to avoid confusion that could reduce the overall clarity and readability of the code base.

Update: Fixed as of commit `4b929c274100a981107e35d40fbf5b57fabc9be4` in pull request #196.