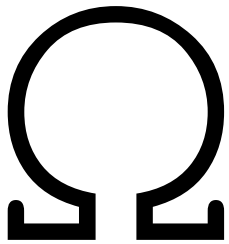
API3DAO
Audit Report

June 15, 2021



Team Omega

Summary

API3DAO has asked Team Omega to audit the contracts that define the behavior of the API3 DAO and the API3 staking pool.

We found 1 issue that we marked as “critical” - these are issues that can lead to the user losing funds. In addition, we classified 6 issues as “medium” and 19 issues as “low”.

An additional 20 issues were classified as “info” - we believe the code would improve if these issues were addressed as well.

Resolution

API3 has addressed the issues raised in this report in https://github.com/api3dao/api3-dao/commit/08ab76af0a1a6461f387075f7b76b47ee4b7e8ea

In this commit, API3 resolved almost all of the issues we raised. One issue classified as medium, two issues we marked as low, and three issues classified as “info”, were left unaddressed for reasons mentioned in the text.

Methods Used

Code Review

We manually inspected the source code to identify potential security flaws.

The contracts were compiled, deployed, and tested in a test environment, both manually and through the test suite provided.

Automatic analysis

We have used several automated analysis tools, including Slither and Remix to detect common potential vulnerabilities. No (true) high severity issues were identified with the automated processes. Some low severity issues, concerning mostly the Solidity pragma version setting and function visibility, were found and we have included them below in the appropriate parts of the report.

Scope of the Audit

The audit concerns the contracts committed here:
https://github.com/api3dao/api3-dao/commit/5fb38bc91ae832f7b94f9630a9d41d15a2c97d27
And specifically the following contracts:

```
packages/api3-voting/contracts
├── Api3Voting.sol
├── interfaces
│   └── IApi3Pool.sol
└── test
    ├── TestImports.sol
    └── mocks
        ├── Api3TokenMock.sol
        └── Api3VotingMock.sol
packages/dao/contracts
├── Api3Template.sol
├── interfaces
│   └── IApi3Pool.sol
└── test
    └── TestImports.sol
packages/pool/contracts
├── Api3Pool.sol
├── ClaimUtils.sol
├── DelegationUtils.sol
├── GetterUtils.sol
├── RewardUtils.sol
├── StakeUtils.sol
├── StateUtils.sol
├── TimelockUtils.sol
├── TransferUtils.sol
├── interfaces
│   ├── IApi3Pool.sol
│   ├── IClaimUtils.sol
│   ├── IDelegationUtils.sol
│   ├── IGetterUtils.sol
│   ├── IRewardUtils.sol
│   ├── IStakeUtils.sol
│   ├── IStateUtils.sol
│   ├── ITimelockUtils.sol
│   └── ITransferUtils.sol
└── mock
    ├── MockApi3Staker.sol
    └── MockApi3Voting.sol
```

Findings

General

Before listing the issues that we found, we start with a general observation: there are a number of related issues having to do with the way snapshots are implemented, which should perhaps be addressed in a systematic way. There are two different implementations of the checkpoint logic in the code - one that is used to keep track of the number of staked tokens, and one used to keep track of delegations (that create a checkpoint for each proposal). Both make use of a linear search algorithm when inspecting the snapshot history. In addition, a rudimentary 1-block history is kept for the totalSupply in addition to a weekly history each time rewards are minted. These different implementations make the code hard to read and understand (as witnessed by several of the issues mentioned below), while the choice for a linear search algorithm that quickly becomes expensive necessitates the introduction of a number of artificial limits (like having to limit the proposal time to 1 week) that add more complexity.

We recommend to consider the option to unify the checkpoint logics, perhaps using one of the MiniMe implementations or OpenZeppelin's ERC20Snapshot implementation.

User can vote twice by delegating her token [resolved]

By delegating her tokens, a user can vote twice for the same proposal. The scenario is simple:

- A proposal is created (the attack only works for the most recent proposal)
- The attacker votes for the proposal
- The attacker delegates her tokens to another address that she controls
- The attacker votes with the new address

The cause of this bug is that `updateCheckpointArray()` will overwrite the value of the latest checkpoint - which is the checkpoint that is used to keep track of the amount of shares delegated to that user.
Recommendation: There is a quick fix for this problem by changing the line in `GetterUtils:299` that reads:

```
if (checkpoints[i - 1].fromBlock <= _block)
```

to

```
if (checkpoints[i - 1].fromBlock < _block)
```

and create the snapshot at the time of proposal creation not for the block that came before the proposal, but rather on the current block itself.
However, as there are many other issues related to this particular way of implementing checkpoints, we recommend a more systematic refactor, in which all functions have a clear semantics.

Severity: Critical
Status: Fixed in https://github.com/api3dao/api3-dao/pull/270

User with proposalThreshold amount of tokens can create 2 proposals in each epoch [low, not resolved]

This works as follows:

1. Create a proposal
2. Delegate to a new account
3. Create another proposal from the new account

Recommendation: A relatively straightforward fix is to require that a user has not made a proposal in the last epoch as a condition to delegate her tokens (in other words, rate-limit proposal creation and delegation together to once per epoch, instead of each separately)
Severity: Low
Status: API3 chose to not address this issue - “the proposal limit is a soft one (we only want to avoid spam, double the number of proposals is not actually a problem)“

Define a single interface for API3Pool [resolved]

The repository contains 4 interfaces called `IAPI3Pool.sol`

```
./packages/dao/contracts/interfaces/IApi3Pool.sol
./packages/api3-voting/contracts/interfaces/IApi3Pool.sol
./packages/pool/contracts/interfaces/IApi3Pool.sol
./packages/pool/contracts/auxiliary/interfaces/IApi3Pool.sol
```

This is confusing, and potentially can lead to errors if these packages define different interfaces for the same contract (we did not find any such errors). Also, as this is a monorepo, there is no reason for duplicating these definitions in the different packages.

Recommendation: define a single complete interface of `API3Pool.sol`, and use that.
Severity: Info
Status: The definition was moved to a new interfaces directory

Inconsistent use of percentage precision [resolved]

In `Api3Voting.sol` (just as Aragon’s `Voting.sol`), `1%` is represented as `10^16`, while in the `pool` contracts, `1%` is represented as `10^6`. This inconsistency makes the code harder to understand, and may lead to errors
Recommendation: represent `1%` as `10^16` throughout the code base
Severity: Info

Status: Fixed as per our recommendation

CircleCI does not run coverage scripts [info, not resolved]

The coverage script is not run as part of the circleci continuous integration pipeline. And the script is actually broken in the api3dao package.
Recommendation: add coverage scripts to the circleci configuration
Severity: Info
Status: “Acknowledged. They take too long so won’t be added. The contracts will not be under continuous development so running them manually at this step is not seen as an issue.”

Vulnerabilities in package dependencies [being resolved]

We ran `npm audit` on May 10, 2021, and reports a number of vulnerabilities in package dependencies:

- In api3-dao root: 29 vulnerabilities (3 low, 2 moderate, 24 high)
- In packages/api3-voting: 73 vulnerabilities (32 low, 5 moderate, 36 high)
- In packages/dao: 40 vulnerabilities (31 low, 1 moderate, 8 high)
- In packages/pool: 59 vulnerabilities (20 low, 5 moderate, 34 high)

As far as we can see, none of these issues directly affect the compilation of the contracts. A deployment script was not provided.
Recommendation: upgrade all the vulnerable packages for which fixes are available.
Severity: Low
Status: “Acknowledged. The dependencies are updated yet they are largely not maintained so the issue is not completely resolved.”

Api3-voting

The api3voting.sol contract is a fork from Aragon’s Voting.sol contract. We audited the differences with the original Voting.sol contract

No license field in package.json [resolved]

`Package.json` contains no license field. This is not obliged, but would be good to have for clarity and consistency.
Recommendation: Add the license
Severity: Info
Status: This was fixed in https://github.com/api3dao/api3-dao/pull/278

Coverage throws an error and so the output is is incomplete [resolved]

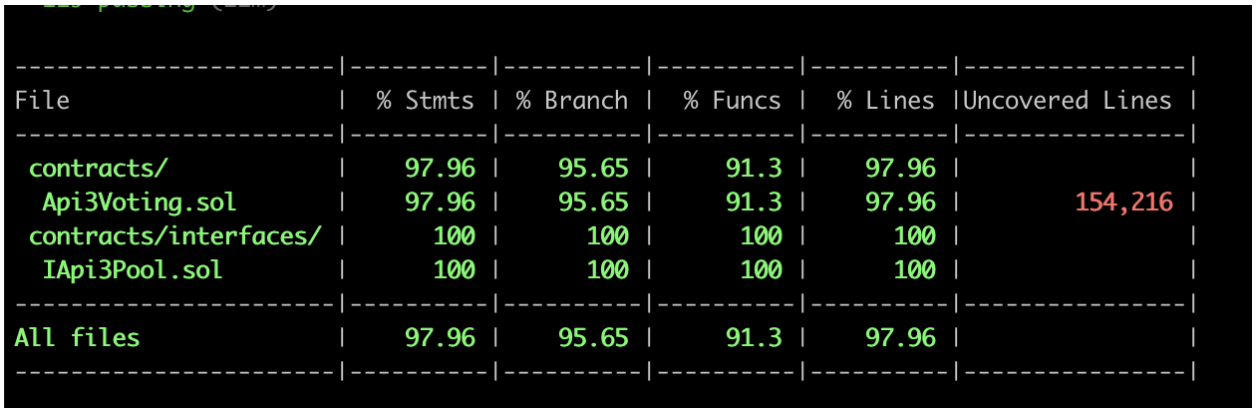
```
----------------------|----------|----------|----------|----------|----------------|
File                  |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------------------|----------|----------|----------|----------|----------------|
 contracts/           |        0 |        0 |        0 |        0 |                |
  Api3Voting.sol      |        0 |        0 |        0 |        0 |... 390,393,394 |
 contracts/interfaces/ |      100 |      100 |      100 |      100 |                |
  IApi3Pool.sol       |      100 |      100 |      100 |      100 |                |
----------------------|----------|----------|----------|----------|----------------|
All files             |        0 |        0 |        0 |        0 |                |
----------------------|----------|----------|----------|----------|----------------|

> Istanbul reports written to ./coverage/ and ./coverage.json
> solidity-coverage cleaning up, shutting down ganache server
Error BDLR700: Artifact for contract "Api3Pool" not found.

For more info go to https://buidler.dev/BDLR700 or run Buidler with --show-stack-traces
npm ERR! code 1
npm ERR! path /Users/jelle/omega/api3-dao/packages/api3-voting
npm ERR! command failed
npm ERR! command sh -c buidler coverage --network coverage
```

Recommendation: Fix the script
Severity: Info
Status: Fixed in https://github.com/api3dao/api3-dao/pull/278/

Incomplete test coverage [resolved]



```
----------------------|----------|----------|----------|----------|----------------|
File                  |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------------------|----------|----------|----------|----------|----------------|
 contracts/           |    97.96 |    95.65 |     91.3 |    97.96 |                |
  Api3Voting.sol      |    97.96 |    95.65 |     91.3 |    97.96 |        154,216 |
 contracts/interfaces/ |      100 |      100 |      100 |      100 |                |
  IApi3Pool.sol       |      100 |      100 |      100 |      100 |                |
----------------------|----------|----------|----------|----------|----------------|
All files             |    97.96 |    95.65 |     91.3 |    97.96 |                |
----------------------|----------|----------|----------|----------|----------------|
```

The functions not tested are `newVote(...)` and `canVote(..)` functions.
Recommendation: test these functions and the deploy script
Severity: Info
Status: This was resolved in https://github.com/api3dao/api3-dao/pull/283/files

Use Safemath for division [resolved]

`Api3Voting.sol:393` does not use `SafeMath`:

```
uint256 computedPct = _value.mul(PCT_BASE) / _total;
```

This is harmless as the line before checks that `_total > 0`
Recommendation: use Safemath's `div` operator, which is a best practice.
Severity: Info
Status: Fixed in https://github.com/api3dao/api3-dao/pull/274

dao

The DAO repository defines an Aragon Template in `Api3Template.sol` that defines functions that can be used to create an Aragon DAO.

The repository is missing a piece of information: there is no code or a settings file that shows with which parameters the API3 dao will be deployed, so we did not analyze that.

Share holders can collectively block insurance claims [low]

As noted in the documentation, stakers have an incentive to try to not have to pay an insurance claim. To avoid that individual stakers temporarily unstake their tokens when an insurance claim is paid out, there is an `unstakeWaitPeriod` that should be longer than the time it takes to evaluate and pay out an insurance claim - perhaps up to a month.

However, stakers may *collectively* be incentivized to avoid paying out a large insurance claim, if they believe that amount withdrawn from their shares outweighs the potential future loss in the token value if the claim is not paid out.

Stakers can block the payout of an insurance claim by accepting a proposal to set the `claimsManagerStatus` to `false`.
Recommendation: set the `voteTime` in the primary voting app to be at least as long as the time it will take to pay out the claim (i.e. to the same value as the `unstakeWaitPeriod`)

It should be noted that this solution is not available if API3 chooses to resolve this issue by setting `voteTime` to be shorter than a week
Severity: Low
Status: The team acknowledged they are aware of this behaviour and write "Shareholders can only block claim payouts by passing a proposal through the primary voting app, which requires 50% quorum. This is intended and desirable."

It is not clear which license applies [resolved]

The `LICENSE` file at the root of the repository contains the MIT license, but `packages/dao/package.json` contains the line `"license": "GPL-3.0-or-later",`
It is not clear which license applies
Recommendation: make clear what license this software is published under
Severity: Low
Status: This was resolved in https://github.com/api3dao/api3-dao/pull/278

setDAOApps is callable by anyone [resolved]

The function `setDAOApps(..)` can be called by anyone. This may present a security risk during the deployment procedure: if `pool.setDAOApps` is called in a separate transaction (as in `test/api3template.js`), an attacker could front-run `setDAOApps` after the pool has been deployed.
Recommendation: include a call to `setDAOApps` as part of the procedure in the factory contract at `Api3Template.newInstance`, so it cannot be front-run.
Severity: Low
Status: This was addressed in https://github.com/api3dao/api3-dao/pull/261/ - `setDAOApps` can now only be called by the deployer or by the MainAgent.

Discrepancies in permissions between documentation and code [resolved]

The documentation in the `packages/dao/README.md` list a number of permissions that are not set accordingly in the code.

App	**Permission**	**Grantee**	**Manager**
MainAgent	TRANSFER	NULL	NULL
SecondaryAgent	TRANSFER	NULL	NULL
Vault(MainAgent)	TRANSFER	MainVoting	MainVoting
Vault(SecondaryAgent)	TRANSFER	SecondaryVoting	SecondaryVoting

We assume that by “Vault(SecondaryAgent)” what is meant is simply the SecondaryAgent itself - although that renders the table inconsistent.
We have confirmed this by checking the DAO deployed in the api3Template.js test, and found some discrepancies. the actual settings we found are the following:

App	Permission	Grantee	Manager
MainAgent	TRANSFER	MainVoting	MainVoting
SecondaryAgent	TRANSFER	MainVoting	MainVoting
Vault(MainAgent)	TRANSFER	MainVoting	MainVoting
Vault(SecondaryAgent)	TRANSFER	SecondaryVoting	MainVoting

Recommendation: Fix the documentation. And add tests for all permissions described in the doc
Severity: Low
Status: This was addressed in https://github.com/api3dao/api3-dao/pull/278 and https://github.com/api3dao/api3-dao/pull/286

newInstance should be external [resolved]

`api3Template.sol:44` The `newInstance` function can be defined external, to save some gas costs
Severity: Low
Status: This was fixed in https://github.com/api3dao/api3-dao/pull/274

Misleading error message [resolved]

`api3Template.sol:52` reads:

```
require(_api3Pool != address(0), "Invalid API3 Api3Voting Rights");
```

The error message would be more correct if it said "_api3Pool address should not be null"
Recommendation: fix the error message
Severity: Low
Status: This was fixed in https://github.com/api3dao/api3-dao/pull/277 and now reads "`API3_INVALID_POOL_ADDRESS`"

Deployment script is not tested [resolved]

The `deploy.js` script, which plays an essential part in the DAO deployment, is not tested
Recommendation: write a test for this script
Severity: Info
Status: There are no formal tests for this script, but there are now detailed instructions on how to verify the deployment in

https://github.com/api3dao/api3-dao/tree/8d8aa794e34780450594349757c45a061e5d3151/packages/dao#verifying-the-dao-deployment

No coverage script [resolved]

There is no coverage script installed, so we have no reports for test coverage.
Recommendation: include a coverage script
Severity: Info
Status: Although a coverage script is not provided, we consider this issue resolved, as API3 writes: “It is difficult to implement test coverage reporting due to the outdated dependencies but it’s manually verified that the contract is fully covered.”

Pool - general remarks

Voting time must be shorter than epoch length [resolved]

If the voting time is longer than the epoch length, token holders that have 1% of the tokens can block any other user from voting on a proposal that is over EPOCH_LENGTH old.

As noted in another issue, `balanceOfAt` may throw an error when called with older block numbers. Specifically, if more than `MAX_INTERACTION_FREQUENCY` proposals are created after a given `blockNumberN`, and the user received a delegation after each proposal, she will be locked out from voting for any proposal created before `blockNumberN`

After a proposal P was created, the attacker divides her tokens over 10 different accounts: each account receives 0.1% of the tokens. For each account, the attacker creates a proposal, and then calls `delegateVotingPower` with the address of the victim. This will created a new checkpoint record in the victim’s `delegatedTo` array

She then waits a week, until the epoch has passed, and repeats the same scenario.

At this point, calling `balanceOfAt` for the block number at which P was created will throw an error, and so the victim will not be able to vote for P anymore.

Note that this attack can be repeated to apply to any number of victims without further costs (except gas costs)

Note also that the attack can be executed with less than 1% of the tokens if there are proposals that are being generated by other users. In a very active DAO, this scenario can happen naturally.

Recommendation: There are various ways to avoid this problem, which is caused by the somewhat arbitrary limit imposed in the search. We recommend using a binary search to search in the checkpoint history, and remove the hard limits.
Another solution, proposed by API3, is to limit the voteTime (i.e. the time within which a proposal needs to be decided) to 1 week.
Severity: Low
Status: This was addressed in two ways. In https://github.com/api3dao/api3-dao/pull/274 setting the voting time to be always equal to the pool's `EPOCH_LENGTHi`. In addition, the new implementation of `balanceOfAt` does not throw an error anymore if called with older block numbers.

Token holders with 2% of the tokens can block any user from being delegated to

There is an attack scenario in which a user that has 2% tokens can lock out a any user from being delegated to for the length of an next epoch

The attacks work like this: at block N, the attacker splits her tokens over 20 accounts with each 0.1% of the tokens (which is the threshold to create proposals). She then creates a proposal, and, with a fresh account, calls `delegateVotingPower` to the victim (as we note elsewhere, any account can call delegateVotingPower). She repeats this 19 times. Finally she creates another proposal.

At this point, any new attempt to call `delegateVotingPower` to the victim will fall for the length of the epoch.

Note that this attack can be extended to apply to any number of victims without additional costs (except for gas costs).
Severity: Medium
Status: Resolved in https://github.com/api3dao/api3-dao/pull/254 and https://github.com/api3dao/api3-dao/pull/255

Use Solidity version 0.8.4 [resolved]

The solidity version for the contracts in the pool package is 0.8.2. Version 0.8.3 contains a bug fix that is classified as "medium", and version 0.8.4 a bug fix classified as "very low".
Recommendation: upgrade to 0.8.4
Severity: Medium
Status: Fixed in https://github.com/api3dao/api3-dao/pull/278

Outdated version of OpenZeppelin (general) [info, not resolved]

OpenZeppelin 3.4.1 is used but 4.0.0 could already be used instead (also compiler version is 0.8.0 instead of version <0.8.0)
Severity: Info

Status: API3 writes “Acknowledged, this is because the Api3Token.sol Solidity version is 0.6.12, which is not supported by more recent OpenZeppelin package versions.”

Organization of code is not helpful [info, not resolved]

The main contract Api3pool is at the bottom of a long linear inheritance chain:

`Api3Pool → TimeLockUtils → ClaimUtils → StakeUtils → TransferUtils → DelegationUtils → RewardUtils → GetterUtils → StateUtils`

Organizing code into independent functional units that can be developed and tested in isolation is a useful pattern. But this is not the pattern that is used here.
For example logic related to “claiming” is now spread over 4 different files (`ClaimUtils` contains the logic, `IClaimUtils` event definitions, `StateUtils` the state variables related to claiming, and `RewardUtils` handles the payment of the claim. This makes the code complex and hard to read and understand.
Recommendation: Ideally, the code would be re-organized into functional units. But this would be a fairly large refactor, and so it is probably not worth the risk
Severity: Info
Status: This was considered to be too much of a refactor, and the code organization was kept as is

Unfair allocation of gas costs for paying rewards (general) [resolved]

User functions such as `delegateVotingPower` and `deposit` call the `payReward` function. This means that only the first user in an epoch will bear the higher gas cost of it, which makes the gas costs for these operations unpredictable and unfair.
Recommendation: create a bot to call the `payReward` function at the start of each EPOCH, so that the problem is transparent to the users; or offer incentives to call the payReward.
Severity: Low
Status: This was acknowledged. API3 may implement a bot

Orphaned file Api3Token.sol (general) [resolved]

There is an empty file in `pool/Api3Token.sol`
Recommendation: remove the file
Severity: Info
Status: Fixed

Underspecified error messages [resolved]

Error messages are often very generic, and do not give any indication of the contract in which the error occurred. Failing transactions are notoriously hard to debug, so the messages should be as specific as possible. This holds specifically for:

- `ERROR_UNAUTHORIZED` simply reads “Unauthorized”, and is used 13 times. It does not specify which authorization is missing. Also, it is not used consistently as authorization error (for example it is used in TimeLockUtils when values are out of range)
- `ERROR_VALUE` reads “Invalid Value”, a phrase that provides no information about what variable has an invalid value. It is used 13 times.
- `ERROR_ADDRESS` reads “Invalid Address”, which does not specify which address is valid, or what it’s invalidity consists of (it is used two times, both in cases where an address is not provided)

Recommendation: Write more specific error messages. Also consider prefixing each error message with a string such as `API3DAO:` so that authors of contracts that will interact with API3DAO can easily identify the source of the error.
Severity: Info
Status: Fixed in https://github.com/api3dao/api3-dao/pull/277

Pool - GetterUtils

balanceOfAt(.., blockNumber) does not return the balance at blockNumber [resolved]

`GetterUtils:15` The result of `balanceOfAt()` function is partially based on checkpoints that are taken when a proposal is created. This means that `balanceOfAt(address, blockNumberN)` will **not** report the voting power at `blockNumberN`, instead it returns a value that is a combination between the balance of the user at that block together with all the delegations it received (or delegated to) the first checkpoint that was created after `blockNumberN`.

In addition, calls to `balanceOfAt` may throw errors in the following cases:

1. Because `balanceOfAt` calls `userReceivedDelegationAt`, the function will throw an error if the user received over `MAX_INTERACTION_FREQUENCY` delegations (from other users) after `blockNumberN`.
2. Because `balanceOfAt` is calculated on the basis of a linear search in 3 lists of checkpoints: `delegates`, `shares` and `delegatedTo`, which are all open-ended, it will run out of gas if these lists become very long and the `blockNumberN` is further in the past

There are several problems here:

1. The documentation is consistently misleading (for example, on `GetterUtils:9` it says `Called to get the voting power of a user at a specific block)`
2. The fact that `balanceOfAt` behaves differently from the MiniMe standard means that it cannot be used as a source for votingPower in other contexts. Specifically, the contract may not be usable if API3 decides to change their decision logic contract, and may present a problem when integrating a system such as snapshot for polling, and a potential pitfall when implementing UI systems.

Note that the similar observations can be made for `userDelegateAt`, `userReceivedDelegationAt`. Each of these functions has confusing doc strings.
Recommendation: We recommend fixing the documentation, and make it very clear that although the contract implements the MiniMe *interface*, it does not implement the MiniMe *semantics*. We also recommend to rename `userDelegateAt`, `userReceivedDelegationAt` and give these functions names that correspond better to their semantics.
Severity: Low
Status: The `balanceOfAt` function was renamed to `userVotingPowerAt and userReceivedDelegationAt` was renamed to `delegatedToUserAt.`
These functions were refactored - a checkpoint is written on each update, and a standard binary search algorithm based on the MiniMe algorithm.

Pool - ClaimUtils

This contract is quite straightforward, and we did not find any issues.

Pool - DelegationUtils

Wrong error message when delegating twice in the same epoch (delegationUtils) [resolved]

`DelegationUtils:30` Should be `ERROR_FREQUENCY` instead of `ERROR_UNAUTHORIZED`
Recommendation: Change the error message
Severity: Low
Status: This was fixed

Missing event [resolved]

`DelegationUtils:38` performs undelegate from the current delegated account, but does not emit the `Undelegated` event, which could cause miscalculations on subgraph level.
Recommendation: add the event

Severity: Low
Status: The event was added

Any account can write a record in the delegateTo [resolved]

`DelegationUtils:8ff` , in `delegateVotingPower`, there is no check that the delegator has any tokens to delegate. This allows any account to write a new record to the `delegatedTo` array, thereby making calls to `balanceOfAt` slightly more expensive. It also makes it easier for an attacker to “fill up” the `delegatedTo` array and block the user from being further delegated to.
Recommendation: require that the user has some tokens to delegate
Severity: Low
Status: A check was added in https://github.com/api3dao/api3-dao/pull/277

Code skips error condition in updateDelegatedVotingPower [resolved]

`DelegationUtils:117` , in `updateDelegatedVotingPower`, checks if the caller is not trying to subtract more from the delegation balance than is available, and if it does, sets the balance to 0.
Because the function is internal, if this happens, this must be an error in the calling function, and so should not be skipped over silently as is currently happening.
Recommendation: do not check for `currentlyDelegatedTo > shares` at all, as this check at best is just a waste of gas, and at most hides a possibly important bug in the calling function.
Severity: Low
Status: This was addressed in https://github.com/api3dao/api3-dao/issues/265

Error in documentation of updateDelegatedVotingPower [resolved]

In `DelegationUtils:95,` it reads that the

```
User shares only get updated while staking, scheduling unstake,
or unstaking
```

This is not strictly true, as the function is not called when scheduling an unstake, but only when the unstake actually has happened
Recommendation: Fix the documentation
Severity: Info
Status: The documentation was updated

Pool - RewardUtils

payReward() mints reward of the *previous* epoch, not the current one [resolved]

`RewardUtils:13` In each epoch, the first time the `payReward()` function is called ,it will update the APR (on the basis of the current stake percentage) and mint the rewards.

The documentation says this function is called to "`pay the reward for the current epoch`", but this is not quite true. Instead, the function is called at the first interaction with the pool in a new epoch, to pay the staking rewards of the *previous* epoch. To be precise: `payReward` is called when the first user stakes in epoch `N`, the users that have shares in epoch `N-1` will get rewarded, but none of the users that stake in epoch `N`.

This is a documentation issue - but the see the following related points

Recommendation: fix the documentation. Also, `mintReward` may better describe what the function is doing.
Severity: Info
Status: The function was renamed to `mintReward` and the documentation was updated

The APR is determined at the end of each epoch, instead of at the beginning [resolved]

`RewardUtils:24` The APR is calculated in line 24, before the reward has been paid out. This makes it impossible for a user to know how high her reward will be in the current epoch, because it depends on how many other stakers will join in this epoch. Specifically, the value of `currentAPR` is ***not*** the "`APR that will be paid next epoch`" (as the doc string claims in `StateUtils:162`) but rather "the APR that stakers have received in the previous epoch".

It is possible that this can be considered unfair by stakers that might stake at the beginning of an epoch attracted by a high "currentAPR", which they have no guarantee of getting. It also makes it harder for new stakers (or the UI) to evaluate their rewards

Recommendation: Consider calling `updateCurrentAPR()` *after* paying the reward (i.e. move it to the end of the `payReward` function) so that the stakers get a predictable APR payout. This in a sense moves the risk from new stakers (who miss out on a higher APR than expected because many stakers join during an epoch) to the token holders in general (who may pay "too much" of a reward if many stakers joined during an epoch).
Another solution, which sidesteps the problem of deciding to calculate the APR before or after the payment of the rewards, to pay the rewards and re-calculate the APR on each stake event instead of only once per epoch (as is also suggested in the next point)
Of course, it is also possible to keep the logic as-is. In that case, we recommend renaming the `currentAPR` variable to a less misleading name, and fixing the documentation.

Severity: Low
Status: This was addressed in https://github.com/api3dao/api3-dao/pull/271, where the call to `updateCurrentAPR` was moved to the end of function as we recommended

Rewards are not calculated on a continuous basis, and there is no continuous relation between staking period and staking rewards [medium, not resolved]

`RewardUtils:13` Rewards are being paid out once every epoch, on the basis of the user's stake at the end of that epoch. This means that a user that stakes on day 1 of an epoch will get the same rewards as a user that stakes on day 7.

For example, a user that enters on day 7 of epoch N and exits on day 1 of epoch N+2 (i.e. stakes her tokens for 8 days) will get exactly the same rewards as a user that enters on day 1 of that epoch N and exits on day 7 of epoch N+2 (i.e. stakes her tokens for 20 days). This may be considered unfair.

In addition, this logic creates incentives to users to stake late in an epoch and unstake early. There are risks associated with staking tokens (e.g. the insurance claims) and there will certainly be opportunity costs (as there will be, almost certainly, other farming opportunities for the API3 token), so users will have incentives to stake late and unstake early. In particular, a user may decide to unstake on day 1 of an epoch and stake again on day 7 of the following, thereby getting the full rewards for both these epochs but at the same time having the token freely available in the intermediate 14 days. With a wait period for unstaking of 1 epoch, this will allow a user to effectively stake only 33% of the time but still get 100% of the staking rewards.

Recommendation: We would recommend refactoring the `payReward()` function and minting the reward "continuously", i.e. each time a user stakes, rather than just once per epoch. This is more in line also with other existing staking contracts.
Severity: Medium
Status: This issue was acknowledged by API3 - it was agreed that implementing our recommendation was not feasible, as it would make the calculation of the vested staking rewards impossible.

The APR calculation is volatile [resolved]

`RewardUtils:45ff` calculates how much the APR should be updated to incentivize (or disincentivize) users to reach the `stakeTarget`. Specifically, the APR in the new epoch as a percentage of the APR in the previous epoch.

This choice seems arbitrary. We do not know much about the relation between the APR and the totalStakeTarget (we assume a higher APR leads to a higher totalStake, but that is pretty much all we know). In particular, there is no reason to think that the absolute size of the previous APR is an indicator of how far the APR is off from the "ideal" APR.

By way of an example, suppose the relation between APR and staking percentage is given by `totalStakePercentage = apr * 2` - i.e. if the apr doubles, then so will the amount of tokens staked. Assume that `minApr` = 1% (which is the default value) and that the `stakeTarget` is 20%. To reach that target, the APR should be set to 10%.

If we start, in epoch 0, with a `totalStakePercentage` of 40% (so the `apr` is at 20%), this will happen

epoch	APR	totalStakePercentage
0	20%	40%
1	1%	2%
2	1.9%	3.8%
3	3.439%	6.878%
4	5.6953279%	11.3906558%
5	8.146979811%	16.29395962%
6	9.656631618%	19.31326324%
7	9.988209815%	19.97641963%

As you can see, the APR overshoots the target in epoch 1, setting the APR to the minimal value, and then it takes 6 epochs before it get within 1 percentage point of the ideal rate.

Although the documentation does not specify under what constraints the APR should change, the table above shows that the APR change can be extremely volatile, which is a disincentive for users to stake. This is surely not what is intended.

Recommendation: An alternative, and more simple, implementation is Livepeer's algorithm. Their use case (target a staking percentage goal by setting farming rewards) is very similar, and their algorithm is more straightforward and has been tested for over 3 years.
It uses a simple absolute rate change.
https://github.com/livepeer/protocol/blob/streamflow/contracts/token/Minter.sol#L207
We recommend changing the APR in absolute staps - or a fixed percentage point. This will not only diminish the volatility of the APR changes, but also simplify the code (lower risk of bugs) and probably save some gas costs.
Severity: Medium
Status: API3 implemented a simple Livepeer style calculation in
https://github.com/api3dao/api3-dao/pull/271

If APR is 0%, it will always remain so [resolved]

If the APR is 0%, the new APR will always remain 0%.

This is a limit case that will happen at epoch N+1 with the current APR calculation, if at epoch N

(a) `minApr` is set to `0` and
(b) `totalStakePercentage > 2 * stakeTarget`.

The DAO can recover in the next epoch by explicitly setting `minApr` to be above 0% as an emergency measure. But that may be unwanted as well as the DAO may want to end inflation altogether if the total staked amount is above the target
Recommendation: As the APR calculation is problematic for other reasons as well, we recommend changing the calculation as described in the previous point
Severity: Low
Status: this was fixed in https://github.com/api3dao/api3-dao/pull/271

Only the rewards of the currently active epoch can be paid [resolved]

The `payRewards` function will pay the rewards for the current epoch only. There is no way to pay the rewards of a past epoch. This means that if for some reason, this function is not called during the current epoch, users will not receive any rewards.
Recommendation: Add a `payRewards(epoch)` function that allows for past and present reward payouts
Severity: Info
Status: This is a known limitation and mentioned in the README.

The size of the reward depends on the values REWARD_VESTING_PERIOD and EPOCH_LENGTH [resolved]

`RewardUtils:25` calculates the reward as follows:

```
uint256 rewardAmount = totalStake * currentApr / REWARD_VESTING_PERIOD
/ HUNDRED_PERCENT;
```

Because `REWARD_VESTING_PERIOD` happens to be set to 52, this amounts to the reward per week, and because the reward is paid once every `EPOCH_LENGTH`, which happens to be set to 7 days, so this works out correctly as approximately an *Annual* percentage rate.

If these constants are changed in future iterations, the “APR” would not be “annual”. It would be more robust to change this to:

```
uint256 rewardAmount = totalStake * currentApr * EPOCH_LENGTH /
ONE_YEAR_IN_SECONDS / HUNDRED_PERCENT;
```

Recommendation: Either change the calculation as indicated, or, alternatively, remove any ambiguity and rename `EPOCH_LENGTH` to `ONE_WEEK`, and `REWARD_VESTING_PERIOD` to `WEEKS_IN_A_YEAR`
Severity: Low
Status: Fixed in https://github.com/api3dao/api3-dao/pull/271

epochIndexOfLastRewardPayment can be updated by anyone [resolved]

`RewardUtils.sol:22` updates the `epochIndexOfLastRewardPayment` regardless of whether `isMinterStatus(this)` is true. This means if in an (unlikely, but possible) scenario where the contract is not the minter at the time of calling this function, but later it does become minter, it will not be able to pay the Reward.
Recommendation: reset the `epochIndexOfLastRewardPayment` only if `isMinterStatus(this)` is true (i.e. switch lines 38 and 39)
Severity: Info
Status: This is considered desired behavior by API3

Duplicate variable name [resolved]

`RewardUtils.sol:34` the variable `userDelegate` has the same name as the function `userDelegate`. This is confusing.
Recommendation: choose a fresh name for the variable
Severity: Info
Status: This was fixed in https://github.com/api3dao/api3-dao/pull/277

Pool - StateUtils

publishSpecsUrl is writable by owner at any time [resolved]

`StateUtils:398`, the function `publishSpecsUrl` can be called by the owner of a proposal at any time.
It is not clear what this function will be used for, but it seems risky if the “specs” of a proposal may change at any time.
Recommendation: Require that the value has not been set previously
Severity: Low
Status: This function was removed in https://github.com/api3dao/api3-dao/pull/277

Redundant check [resolved]

`StateUtils:284` checks if `_stakeTarget >= 0`. This can be omitted, as this is always true
Recommendation: remove the check
Severity: Info

Status: This was resolved in https://github.com/api3dao/api3-dao/pull/266

Pool - StakeUtils

Scheduling to unstake has no costs [resolved]

In StakeUtils.sol, `scheduleUnstake` can be called without any penalty - in particular, if the user scheduled to unstake, she will continue to receive rewards. So (disregarding gas costs) a rational staker should schedule an unstake for each epoch, so she can exit at any point. This makes the unstake scheduling sort of pointless.
Recommendation: Make it so that the user can only schedule a single unstake event at a time. In addition, if a user schedules an unstake, she loses the right to further rewards, and her right to vote.
Severity: Medium
Status: Our recommendation was implemented in https://github.com/api3dao/api3-dao/pull/272

Pool - TransferUtils

Withdrawing tokens for active user can become very expensive, and the user may be locked out from withdrawing her tokens [resolved]

`GetterUtils:241, getUserLocked` loops over all `user.shares` checkpoints in the past year. A new element is pushed on the `user.shares` array each time a user stakes or unstakes. There is no limit on the size of that array, so calling the `getUserLocked` function can become very expensive, and if the array is too big, it may run out of gas. If a user stakes/unstake 50 times a week, the costs are over 7M gas, which is close to the current gas limit of 12.5M (and is of course terribly expensive)

As `getUserLocked` is called in the `withdraw` function, this means withdrawing may become prohibitively expensive or even impossible. The search space will diminish if the user stops staking and lets time pass.

The README file mentions this problem and says “it is not considered an issue” and suggests that this will only happen in abnormal cases:

> `In case the user bloats this array on purpose by repeatedly staking 1 (Wei) API3, they may manage to lock themselves out of voting/withdrawing.`

```
Since this will only happen voluntarily and will get resolved
automatically simply by not staking/unstaking for a while, it is
not considered as an issue.
```

We do not think this addresses the issue.
First of all, the problem does not only regard “users that stake 1 wei”, but rather all power users. For example, an active on-chain staking pool such a yearn.finance vault, or a staking bot may want to interact intensively with staking contract, perhaps also many times in a day. In addition, for a public on-chain staking protocol, simply “not staking for a while” may be not an available option.
In addition, the problem not only concerns users being locked out, but high gas costs in general. Here are some examples:

Staking activity	Gas cost to withdraw tokens after 52 weeks
Once	`398K`
1 time a week	`464K`
10 times a week	`1.829K`
50 times a week	`7,178K`

Compare these values with a simple ether transfer (21K), an ERC20 token transfer (about 60K) and the current block size limit (12.5M)

Recommendation:
Because rewards are paid only once per epoch, it does not seem to be necessary to store a new checkpoint in `user.shares` on each stake/unstake event. Instead, one can use a logic such as that implemented in `updateCheckpointArray`, which will limit allow a refactor of the iteration in `getUserLocked` to use at most 52 iterations (and in most cases much less)

A different approach, that would require a larger change: In the current implementation, locked rewards are minted directly to the pool where they are indistinguishable from the deposited tokens and the rewards that are available for withdrawal. This forces the developer to recalculate, on each withdrawal, which of the tokens are still locked, which is expensive. Consider turning this logic around, and instead use a more “pull” style logic where rewards are stored separately from the active pool, and users must claim their rewards before they can withdraw them.

Severity: Medium

Status: This was addressed in https://github.com/api3dao/api3-dao/pull/254 with the addition of the addition of functions called `precalculateUserLocked` and `withdrawPrecalculated`

Check return value from transferFrom [resolved]

`TransferUtils:24` on this line `api3Token.transferFrom` is called. The ERC20 specification defines that `transferFrom` returns `false` if it fails (i.e. does not necessarily throw an error). Even if `api3token` uses the OpenZeppelin implementation, which does throw an error, it is best practice to check if the return value is indeed true.
Recommendation: check return value of `transferFrom`
Severity: Info
Status: This was fixed

Unused argument in depositAndStake [resolved]

In `StakeUtils:43` the argument `userAddress` is not necessary, as it is always equal to `msg.sender`
Recommendation: remove the unused argument
Severity: Low
Status: Resolved

Severity definitions

Critical	Vulnerabilities that can lead to loss of assets or data manipulations.
Medium	Vulnerabilities that are essential to fix, but that do not lead to assets loss or data manipulations
Low	Issues that do not represent direct exploit, such as poor implementations, deviations from best practice, high gas costs, etc
Info	Matters of opinion