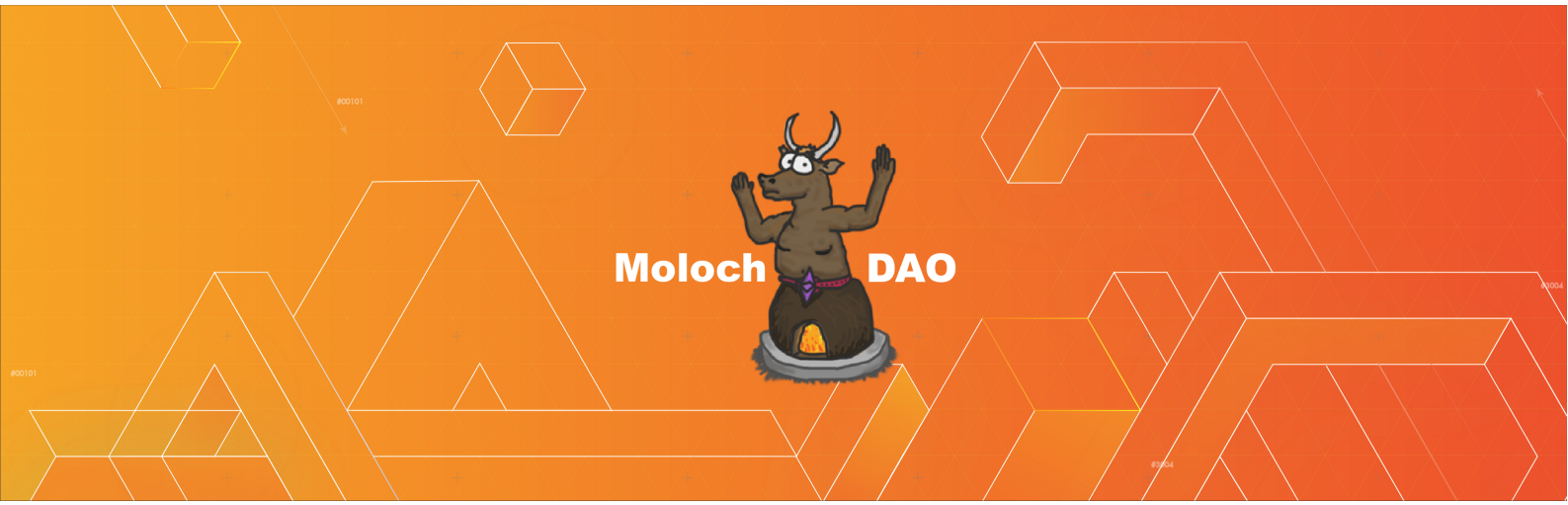
Moloch DAO Audit Report

Moloch whose mind is pure machinery! Moloch whose blood is running money! Moloch whose fingers are ten armies!

Nomic Labs Follow
Feb 14, 2019 · 8 min read



Nomic Labs contributed a smart contract security audit to the Moloch DAO project. The audited version is 5e82f17d5efa894976e50701cf3fa2f88d2372b5, and we found no significant security issues.

. . .

Audit results

Low severity vulnerabilities

[MOL-L01] Misbehaving members can make proposal submissions fail

Submitting a proposal requires two successful transactions to work. A first one to approve the Moloch DAO to use `transferFrom` to collect the `proposalDeposit` from the proposer, and a second one to call `submitProposal`.

A member M1 could be monitoring the network to detect when a member M2 approved the Moloch contract and sent a transaction to `submitProposal`. M1 could then submit another proposal, with M2 as `applicant`, causing M2's proposal submission to fail.

If this happened M2 could call `abort` to get their funds back and try to resubmit the proposal.

This attack would only cost `processingReward` to M1, plus the time value of having `proposalDeposit` funds locked. In concrete terms, M1 would need to lock around 1000 USD for about two weeks, losing around 10 USD. This cost is potentially too high for realistic prolonged attacks, but such an attack could still be used to create conflict on controversial proposals, especially the ones M1 opposes.

Update from Moloch: This is a great find. Existing members frontrunning proposal submission from other members can be used as a griefing vector. However, because this attack can only be launched by existing members we consider it low-risk. If we detect that any member is abusing this, however, we will prioritize launching an upgraded contract with the fix (as proposed in `MOL-L03`).

[MOL-L02] Members that ragequitted can't update their delegate keys

If a member ragequits with all of their shares, they can't update their delegate key. This can lead to unnecessary complications in case of a delegate key being compromised. An example of such a situation is:

1. Member M1 gets proposed and accepted.
2. Member M1 sets D1 as their delegate key.
3. D1 is compromised and used by an attacker to vote.
4. Member M1 detects this and front-runs the vote with a `ragequit` transaction to stop the attack.
5. M1 can't change their delegate key anymore.
6. If M1 wants to be a member again, extra care has to be taken when processing their new application, or the attacker would be able to use D1 again.

While the possibility of these being exploited is low, we recommend splitting the `member` role in two, depending on the number of shares. See `[MOL-005]` and `[MOL-006]`.

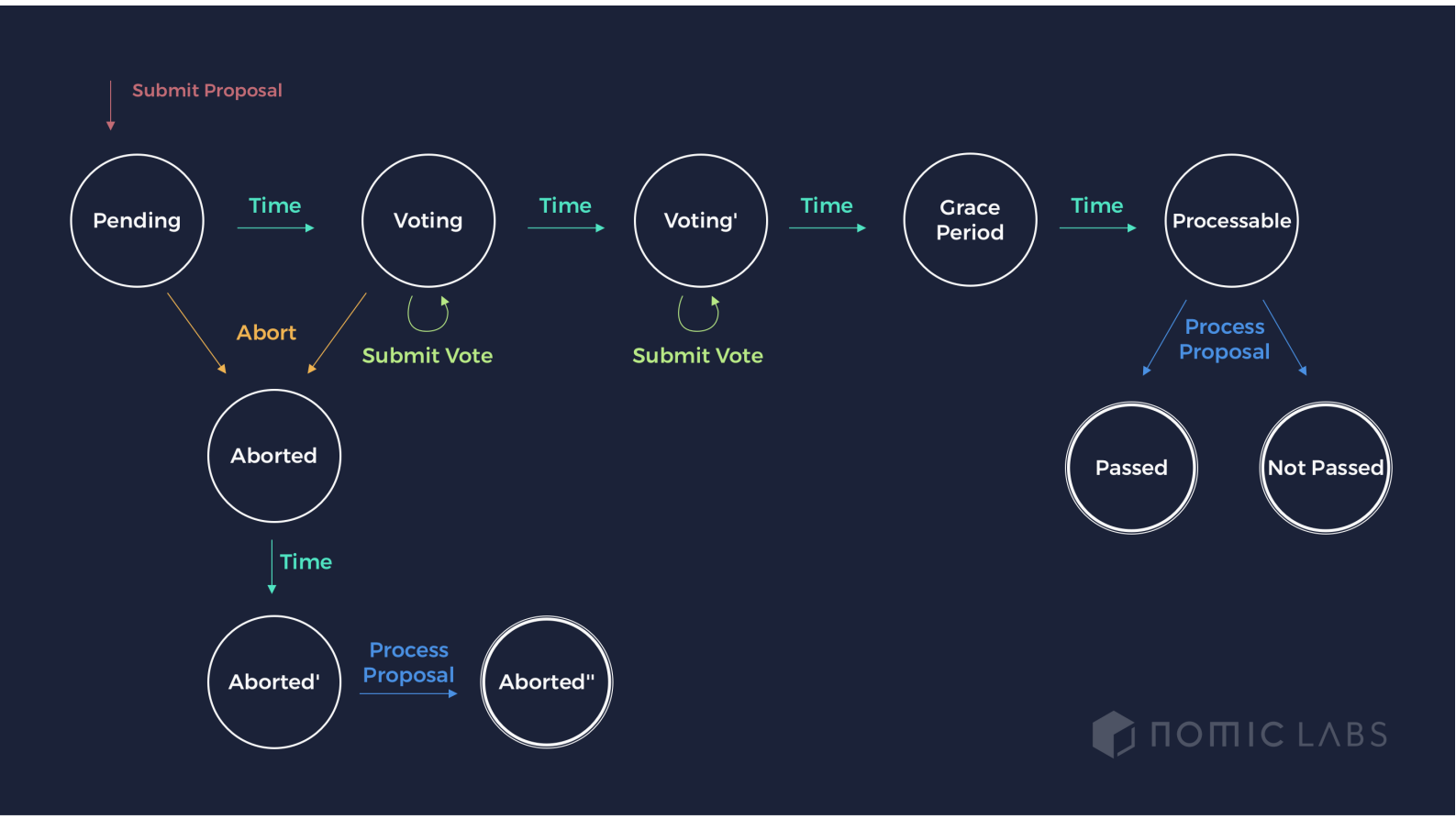
Update from Moloch: If a delegate key is hacked, we think it is likely that a member would first call `updateDelegateKey` from their member address before trying to ragequit (step #4). Further, if the hacker votes YES using the member's delegate key, then the member won't be able to ragequit at all. We did not implement a fix for this.

[MOL-L03] Approving the Moloch DAO to transfer tokens is unsafe

Approving the Moloch DAO to transfer your tokens is, in general, unsafe. Users need to approve tokens to be a proposer or an applicant, but they can end up as the applicant of an unwanted proposal if someone attacks them, as explained in `[MOL-L01]`.

This also has an impact in the UX, as submitting a proposal requires three transactions (2 approvals, 1 `submitProposal` call). This is in contrast to one of the most common UX pattern for approval, which consists of only calling approve once, with `MAX_INT` as value. If someone were to use that pattern, she will be in a vulnerable situation.

Part of this problem has been mitigated by giving applicants the ability to abort their proposals. With this mitigation included, the life cycle of a proposal is shown below.

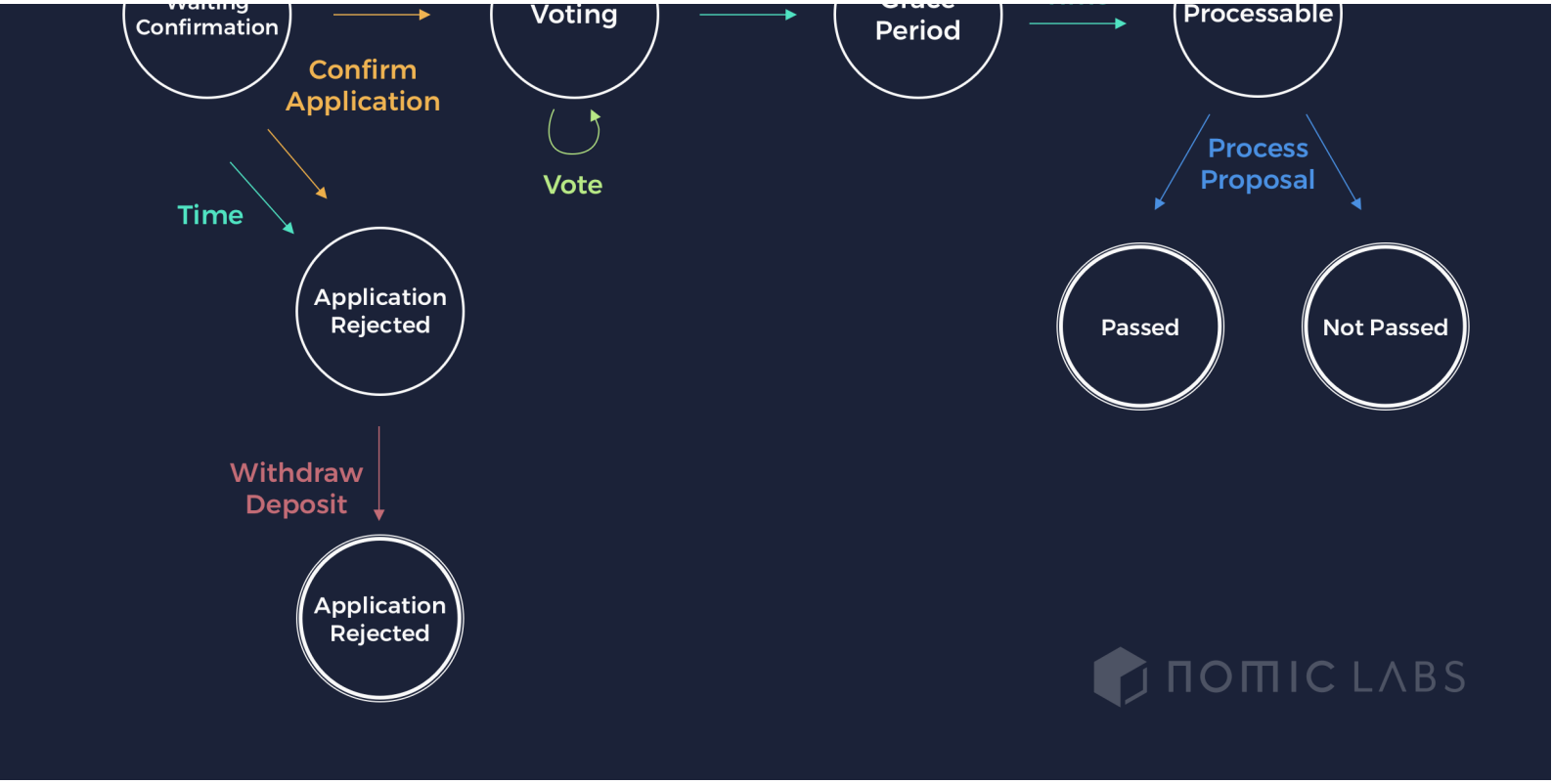


The root of this problem is that `submitProposal` collects the funds of both the `proposer` and the `applicant`.

We propose an alternative design, where applicants have to confirm their application before the proposal gets accepted, and where each user is responsible for sending the transaction that collects their funds.

`Moloch#submitProposal` would use `transferFrom` from the ERC20 contract to collect the `proposalDeposit`, and `confirmApplication` would use it to collect the applicant's tribute.

The alternative life cycle of a proposal would be:



The advantages of this alternative design are:

1. It's a simpler life cycle.
2. Nobody can make a user an applicant without their consent.
3. Users can approve the Moloch DAO just once.
4. It can be implemented in a way that the proposal deposit gets collected from the members, and not their delegates. See `[MOL-007]` below.

Update from Moloch: We agree that this is an important vulnerability and that the proposed alternative design is sound. As stated in `MOL-L02` , in the interest of time we decided not to implement these changes for this version of the contract, but plan to in the future. We did however document this vulnerability prominently in the readme.

Other comments & recommendations

[MOL-O01] GuildBank#withdraw emits Withdrawal on failure

This function should only emit an event when it can successfully call `approvedToken.transfer` , but it does it unconditionally.

This has no real impact in the system, given that it's only called from the Moloch DAO, and with a well behaved ERC20 as `approvedToken` .

Update from Moloch: This actually will not emit the event on failure because the `GuildBank.withdraw` is wrapped in a require when it is called from the Moloch contract. If the token transfer fails, this function will return false, causing the require in Moloch to fail, and the event emission to be reverted.

[MOL-O02] Usage of OpenZeppelin's ERC20 instead of IERC20

Both contracts, `Moloch` and `GuildBank`, use OpenZeppelin's ERC20 as an interface, but that contract is their own implementation of the standard. `IERC20` should be used instead.

Update from Moloch: fixed in commit acbee32057225ca210013e7ba217e37b73cb6f42

[MOL-O03] The dilution bound functionality is confusing

`Moloch#processProposal` imposes a "dilution bound" with the intention to protect members from excessive dilution in case of a mass ragequit.

The mechanism implemented successfully protects the users in such cases, but it's not a bound of the dilution, as it ignores the number of shares requested. We recommend renaming it.

Update from Moloch: We kept this as-is because we couldn't think of a better name. We feel that the current name fits because the parameter bounds the total dilution you would be expected to suffer if the proposal were to pass.

[MOL-O04] The README.md file is outdated

We recommend not duplicating that much information in the README file, especially code, as it gets outdated very easily.

Update from Moloch: Fixed, but we still like keeping code in the README.md. Now that the contracts are done it shouldn't get outdated anymore!

[MOL-O05] Member's `isActive` field has a confusing name

This field seems to imply that the member has shares and can participate in the voting process, but that's not the case. It only signals that the member has been active at any point in time.

Update from Moloch: Fixed. We changed it to `exists`.

[MOL-O06] `onlyMember` and `onlyDelegate` have confusing names

These names can cause confusion, as they check for members with shares, not just any member. An example of confusing behavior because of this is `[MOL-L02]`.

Update from Moloch: We kept it as is. If you no longer have at least 1 share, we don't consider you a member.

[MOL-O07] `Moloch#submitProposal` collects the `proposalDeposit` from `delegate` instead of `proposer`

This function has the following comment:

> *collect proposal deposit from proposer and store it in the Moloch until the proposal is processed*

but collects the deposit from the `msg.sender`, which is the proposer's delegate, which is not necessarily the member.

We recommend documenting this behavior. As an alternative, the deposit could be collected from the actual member, but that would imply using their key more often, as submitting a proposal requires a token approval transaction.

If the design explained in `[MOL-L03]` were to be implemented, this function could collect the funds from the member, without making it less secure.

Update from Moloch: This is true, and we pull the deposit from the delegate instead of the member address to avoid needing extra interactions with the member key. We kept it as is for now, but when we implement the recommendations from MOL-L03 we will reconsider allowing drawing deposit funds from the member address.

[MOL-O08] Period `0` used as default `highestIndexYesVote` can be confusing

When adding a new member, `highestIndexYesVote` is set to `0`. This makes the Moloch behavior somewhat different during the first `votingPeriodLength + gracePeriodLength` periods.

This doesn't affect any member except for the summoner, but we recommend documenting why it's safe.

Update from Moloch: Good catch. Added a note about this in the readme.

[MOL-O09] Duplicated logic for adding members

The constructor and `processProposal` have duplicated logic for adding members. We recommend extracting a function with it.

Update from Moloch: Kept as is in the interest of time.

[MOL-O10] Submitting a `uintVote` greater than two throws an exception

We recommend checking this and reverting with a nicer error message.

Update from Moloch: Fixed.

[MOL-O11] Resetting a member's delegate key in `processProposal` is unexpected

We find this behavior unexpected and somewhat hard to reason about. We evaluated some alternatives and the following proposal to be the easiest to implement without

opening the door to potential denial of service attacks.

We recommend adding an event so that members can know when their delegate keys have been reset.

An interesting alternative to consider is to let addresses be delegate keys of multiple members. This would require an extra parameter in `submitProposal` and `submitVote`, indicating which member is being represented but would simplify reasoning about the system.

Update from Moloch: Kept as is in the interest of time.

[MOL-O12] Overflow checks can be confusing

`Moloch#submitProposal` checks that the number of shares requested doesn't overflow and lock the submissions processing. This is done by computing a bound on the number of shares with `safeMath`, in a way that it will revert the transaction if an overflow happens. This is not very evident, especially because it's done inside a require.

Update: While writing this report, PR #11 was opened, which implements clearer and more exhaustive overflow checks.

[MOL-013] Incorrect error messages

The constructor has the following `require` with an incorrect error message:

```
require(_abortWindow <= _votingPeriodLength, "Moloch::constructor - _abortWindow must be smaller than _votingPeriodLength");
```

`Moloch#abort` has this `require` with an incorrect error message:

```
require(approvedToken.transfer(proposal.applicant, tokensToAbort), "Moloch::processProposal - failing vote token transfer failed");
```

Update from Moloch: Fixed.

[MOL-014] Some `require` are repeated multiple times

Some like `require(proposalIndex < proposalQueue.length, "Moloch::abort - proposal does not exist");` are repeated multiple times. They are mostly array index checks. We recommend creating getters for those array elements.

Update from Moloch: Kept as is in the interest of time.

[MOL-015] State guards are not always clear

Each of the Moloch DAO functions has several requires to validate that they are only called in the correct state of the proposal they refer to. We recommend moving this logic

into boolean functions with clearer semantics and names.

For example, `submitVote` would have a single require checking the return value of `isVotable(proposalIndex)`. Similar functions would be implemented for each or most of the states shown in the diagrams in `[MOL-L03]`.

Update from Moloch: Kept as is in the interest of time.

. . .