SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Nouns
Prepared by:	Sherlock
Lead Security Expert:	hyh
Dates Audited:	March 20 - April 1, 2024
Prepared on:	April 22, 2024

Introduction

This project solves the problem of allowing payers of token streams to create streams and fund them after the fact, i.e. first create the stream, then fund it later when funds are available.

Scope

Repository: nounsDAO/nouns-monorepo

Branch: verbs-poc-client-incentives-noracle

Commit: 5ab971094ea14de9a3bdb13d3222a49e36d87199

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
1	2

Issues not fixed or acknowledged

Medium	High
0	0

Issue H-1: Rewards can be stolen from other proposals and votes by extending auction revenue period with the help of bogus proposals

Source: https://github.com/sherlock-audit/2024-03-nouns-dao-2-judging/issues/46

Found by

hyh

Summary

Attacker controlling more than `proposalThresholdBPS` share of votes can systematically create auxiliary proposals to capture a bigger period of auction revenue as rewards for ones they are affiliated with.

It makes sense for a beneficiary of the currently ended `proposal 0` to front-run any next valid proposal creation with a dummy proposal, so it will be an anchor last proposal for `updateRewardsForProposalWritingAndVoting(anchorProposalId, ...)` call so all the auction revenue before the next valid proposal creation moment will go to `proposal 0`. This will systematically increase their rewards at the expense of other client id owners, i.e. stealing from them.

Vulnerability Detail

The auction reward period can be artificially extended at the expense of other eligible proposals' rewards with the help of automated creation of bogus proposals. That's possible as the auction revenue period is derived from the user supplied last proposal id before proposal eligibility control. I.e. an attacker can front run next valid proposal creation with the creation of the fake proposal in order to maximize the auction revenue period for the proposal they have interest in (i.e. have any affiliation with client id owner of proposal or its votes).

Fake proposal doesn't have to be eligible: it can be heavily voted against, it can be vetoed. The only requirement is that it should have ended (`require(block.number > endBlock, ...)`) as of the time of the reward allocation call. Note that `objectionPeriodEndBlock` is activated only for flipped proposals, which is highly unlikely for bogus ones, so `endBlock = max(proposals[i].endBlock, proposals[i].objectionPeriodEndBlock) = proposals[i].endBlock`, which is known as of time of proposal creation. In general it should be as late as possible, for example front running the next rival reward allocation proposal.

Schematic POC:

1) suppose a `proposal 1` that will drive much attention and votes is known to be published
2) attacker is affiliated to a beneficiary of a `clientId` used in creation of some already passed or almost passed not so popular `proposal 0`
3) attacker front runs `proposal 1` creation with creation of its own `proposal 2`. Both have the same creation time
4) attacker runs `updateRewardsForProposalWritingAndVoting(proposal_2_id, ids)` with `ids` covering all the activity of the yet uncovered period
5) Their `clientId` is getting all the auction revenue prior to `creation_time` undiluted by votes of `proposal 1` (which can be expected to be larger than ones of `proposal 0`)

Attacker has effectively stolen rewards from client id owner of `proposal 1` proposal and votes.

Impact

The attack can be carried deterministically, there are no direct prerequisites, while the total cost is additional proposal creation gas only. The probability of the mentioned setup occurring so the attack will make sense can be estimated as medium as the situation is pretty typical. Auction period manipulation has substantial impact on rewards calculation, so the material impact on the other client id owners' reward revenue is high.

Likelihood: Medium + Impact: High = Severity: High.

Code Snippet

The revenue allocated for reward distribution is based on the creation time of the `t.lastProposal`:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/client-incentives/Rewards.sol#L332-L342

```
         t.firstAuctionIdForRevenue = $.nextProposalRewardFirstAuctionId;
         (uint256 auctionRevenue, uint256 lastAuctionIdForRevenue) =
 ↪  getAuctionRevenue({
             firstNounId: t.firstAuctionIdForRevenue,
>>           endTimestamp: t.lastProposal.creationTimestamp
         });
         $.nextProposalRewardFirstAuctionId = uint32(lastAuctionIdForRevenue) + 1;

         require(auctionRevenue > 0, 'auctionRevenue must be > 0');
```

```
        t.proposalRewardForPeriod = (auctionRevenue *
 ↪  $.params.proposalRewardBps) / 10_000;
        t.votingRewardForPeriod = (auctionRevenue * $.params.votingRewardBps) /
 ↪  10_000;
```

Which is user supplied `lastProposalId` as `proposalDataForRewards` doesn't filter the proposals, returning all the results sequentially:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/client-incentives/Rewards.sol#L319-L330

```
        require(lastProposalId <= nounsDAO.proposalCount(), 'bad
 ↪  lastProposalId');
        require(lastProposalId >= t.nextProposalIdToReward, 'bad
 ↪  lastProposalId');
        require(isSortedAndNoDuplicates(votingClientIds), 'must be sorted &
 ↪  unique');

>>      NounsDAOTypes.ProposalForRewards[] memory proposals =
 ↪  nounsDAO.proposalDataForRewards(
            t.nextProposalIdToReward,
>>          lastProposalId,
            votingClientIds
        );
        $.nextProposalIdToReward = lastProposalId + 1;

>>      t.lastProposal = proposals[proposals.length - 1];
```

I.e. here `t.lastProposal` doesn't have to be eligible, it can be any fake proposal which will be omitted later on in the logic. But since `t.lastProposal.creationTimestamp` is used for `getAuctionRevenue()` it will have an impact of capturing a bigger share of auction profits:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/client-incentives/Rewards.sol#L531-L542

```
    function getAuctionRevenue(
        uint256 firstNounId,
        uint256 endTimestamp
    ) public view returns (uint256 sumRevenue, uint256 lastAuctionId) {
        INounsAuctionHouseV2.Settlement[] memory s =
 ↪  auctionHouse.getSettlementsFromIdtoTimestamp(
            firstNounId,
>>          endTimestamp,
            true
        );
        sumRevenue = sumAuctions(s);
        lastAuctionId = s[s.length - 1].nounId;
    }
```

Allowing the attacker to maximize the auction revenue period for rewards calculation as the whole period from `startId` to `maxId` will be used, while `endTimestamp` was manipulated to be close to the current moment:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/NounsAuctionHouseV2.sol#L452-L480

```
    function getSettlementsFromIdtoTimestamp(
        ...
    ) public view returns (Settlement[] memory settlements) {
        uint256 maxId = auctionStorage.nounId;
        require(startId <= maxId, 'startId too large');
        settlements = new Settlement[](maxId - startId + 1);
        uint256 actualCount = 0;
        SettlementState memory settlementState;
>>      for (uint256 id = startId; id <= maxId; ++id) {
            settlementState = settlementHistory[id];

            if (skipEmptyValues && settlementState.blockTimestamp <= 1) continue;

            // don't include the currently auctioned noun if it hasn't settled
            if ((id == maxId) && (settlementState.blockTimestamp <= 1)) continue;

>>          if (settlementState.blockTimestamp > endTimestamp) break;

            settlements[actualCount] = Settlement({
                blockTimestamp: settlementState.blockTimestamp,
                amount: uint64PriceToUint256(settlementState.amount),
                winner: settlementState.winner,
                nounId: id,
                clientId: settlementState.clientId
            });
            ++actualCount;
        }
```

Tool used

Manual Review

Recommendation

Consider basing the check on the last eligible proposal by filtering them out in `proposalDataForRewards()`, which isn't used outside reward logic:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/governance/NounsDAOProposals.sol#L719-L737

```
    function proposalDataForRewards(
        NounsDAOTypes.Storage storage ds,
        uint256 firstProposalId,
        uint256 lastProposalId,
+       uint16 proposalEligibilityQuorumBps,
        uint32[] calldata votingClientIds
    ) internal view returns (NounsDAOTypes.ProposalForRewards[] memory) {
        require(lastProposalId >= firstProposalId, 'lastProposalId >=
 ↪  firstProposalId');
        uint256 numProposals = lastProposalId - firstProposalId + 1;
        NounsDAOTypes.ProposalForRewards[] memory data = new
 ↪  NounsDAOTypes.ProposalForRewards[](numProposals);

        NounsDAOTypes.Proposal storage proposal;
        uint256 i;
        for (uint256 pid = firstProposalId; pid <= lastProposalId; ++pid) {
            proposal = ds._proposals[pid];
+           if (proposal.canceled || proposals.forVotes < (proposals.totalSupply
 ↪  * proposalEligibilityQuorumBps) / 10_000) continue;

            NounsDAOTypes.ClientVoteData[] memory c = new
 ↪  NounsDAOTypes.ClientVoteData[](votingClientIds.length);
            for (uint256 j; j < votingClientIds.length; ++j) {
                c[j] = proposal.voteClients[votingClientIds[j]];
            }
```

In order to minimize the changes this suggestion is shared with another issues.

This way `getAuctionRevenue()` will be called with the `endTimestamp` based on the last proposal after filtration.

Discussion

eladmallel

We are working on a fix that will better filter out proposals.

We think severity should be lowered to Medium; still thinking the term "loss of funds" is not quite appropriate here, but this criterion for medium issues from

Sherlock docs seems relevant:

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.

While the definitions for high seem less fitting to this issue.

WangSecurity

I believe here the High severity is appropriate since the prerequisites and conditions are not external and can be created by the attacker.

WangSecurity

Comment from LSW:

"the attacker only need to be able to surpass proposalThresholdBPS, which is and will be kept low enough to facilitate governance participation. There are no other ifs here, one can manipulate the auction span straightforwardly."

Under these reasons the report remains High seveirty.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits: https://github.com/nounsDAO/nouns-monorepo/pull/839

dmitriia

Fix looks ok: bogus proposals are now filtered out in `proposalDataForRewards()` as eligibility criteria was moved there.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue H-2: Eligibility of cancelled proposals makes it possible for `proposalEligibilityQuorumBps` controlling actor to create multiple eligible proposals, stealing rewards from all others

Source:
https://github.com/sherlock-audit/2024-03-nouns-dao-2-judging/issues/51

Found by

hyh

Summary

Actor controlling more than `proposalEligibilityQuorumBps` can spam eligible proposals via creation, voting and canceling them, stealing proposal based rewards from all other participants. Since `proposalEligibilityQuorumBps` is set to `10%`, it can require colluding of the several Nouns holders. Spamming here means that this have much shorter turnaround time compared to usual workflow and has a potential of heavily diluting the rewards for all others.

Vulnerability Detail

Suppose an actor is affiliated with `clientId` being `approved` and controls (directly or indirectly via collusion) more than `proposalEligibilityQuorumBps` of the current total supply. They can repeat `(create, vote, cancel)` cycle over time obtaining a significant share of proposal rewards since this will produce eligible proposals substantially faster compared to the normal lifecycle as full `votingPeriod` is omitted.

In order to conceal the manipulation this can be intermixed with other activities: i.e. before most valid proposal's, having target `clientId`, voting and execution there might be some deliberately broken versions of this proposal being posted, then voted on and cancelled like if the inconsistency was just being discovered. This can be done gradually in order to avoid raising red flags and keep `clientId` valid.

Impact

The only prerequisite is some arrangement of Nouns holders in order in exceed `proposalEligibilityQuorumBps` bar. All the rest can be done routinely. The manipulation will not be evident on rewards allocation as `updateRewardsForProposalWritingAndVoting()` will count manipulated proposals automatically.

Increasing the number of eligible proposals has substantial impact on rewards calculation, so the impact on the other client id owners' reward revenue is high.

Likelihood: Medium + Impact: High = Severity: High.

Code Snippet

Cancelled proposals will be deemed valid and will dilute the proposal rewards base:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/client-incentives/Rewards.sol#L352-L368

```
        for (uint256 i; i < proposals.length; ++i) {
            // make sure proposal finished voting
            uint endBlock = max(proposals[i].endBlock,
 ↪  proposals[i].objectionPeriodEndBlock);
>>          require(block.number > endBlock, 'all proposals must be done with
 ↪  voting');

            // skip non eligible proposals
>>          if (proposals[i].forVotes < (proposals[i].totalSupply *
 ↪  proposalEligibilityQuorumBps_) / 10_000) {
                delete proposals[i];
                continue;
            }

            // proposal is eligible for reward
            ++t.numEligibleProposals;

            uint256 votesInProposal = proposals[i].forVotes +
 ↪  proposals[i].againstVotes + proposals[i].abstainVotes;
            t.numEligibleVotes += votesInProposal;
        }
```

Proposals can be cancelled by `proposer` early on (whenever in a non-final state), i.e. right after `proposalUpdatablePeriodInBlocks + votingDelay` passed and proposal becomes `Active`, so it can be voted on and cancelled:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/governance/NounsDAOProposals.sol#L518-L547

```
    function cancel(NounsDAOTypes.Storage storage ds, uint256 proposalId)
 ↪  external {
        NounsDAOTypes.ProposalState proposalState = stateInternal(ds,
 ↪  proposalId);
```

```
        if (
            proposalState == NounsDAOTypes.ProposalState.Canceled ||
            proposalState == NounsDAOTypes.ProposalState.Defeated ||
            proposalState == NounsDAOTypes.ProposalState.Expired ||
            proposalState == NounsDAOTypes.ProposalState.Executed ||
            proposalState == NounsDAOTypes.ProposalState.Vetoed
        ) {
>>          revert CantCancelProposalAtFinalState();
        }

        NounsDAOTypes.Proposal storage proposal = ds._proposals[proposalId];
        address proposer = proposal.proposer;
        NounsTokenLike nouns = ds.nouns;

        uint256 votes = nouns.getPriorVotes(proposer, block.number - 1);
        bool msgSenderIsProposer = proposer == msg.sender;
        address[] memory signers = proposal.signers;
        for (uint256 i = 0; i < signers.length; ++i) {
            msgSenderIsProposer = msgSenderIsProposer || msg.sender ==
↪   signers[i];
            votes += nouns.getPriorVotes(signers[i], block.number - 1);
        }

        require(
>>          msgSenderIsProposer || votes <= proposal.proposalThreshold,
            'NounsDAO::cancel: proposer above threshold'
        );

>>      proposal.canceled = true;
```

After that proposer and other backers can immediately create new proposal since the state of their current one is `Canceled` and the corresponding check is satisfied:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/governance/NounsDAOProposals.sol#L778-L789

```
>>  function checkNoActiveProp(NounsDAOTypes.Storage storage ds, address
↪   proposer) internal view {
        uint256 latestProposalId = ds.latestProposalIds[proposer];
        if (latestProposalId != 0) {
            NounsDAOTypes.ProposalState proposersLatestProposalState =
↪   stateInternal(ds, latestProposalId);
            if (
>>              proposersLatestProposalState ==
↪   NounsDAOTypes.ProposalState.ObjectionPeriod ||
>>              proposersLatestProposalState ==
↪   NounsDAOTypes.ProposalState.Active ||
>>              proposersLatestProposalState ==
↪   NounsDAOTypes.ProposalState.Pending ||
>>              proposersLatestProposalState ==
↪   NounsDAOTypes.ProposalState.Updatable
            ) revert ProposerAlreadyHasALiveProposal();
        }
    }
```

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/governance/NounsDAOProposals.sol#L582-L592

```
    function stateInternal(
        NounsDAOTypes.Storage storage ds,
        uint256 proposalId
    ) internal view returns (NounsDAOTypes.ProposalState) {
        require(ds.proposalCount >= proposalId, 'NounsDAO::state: invalid
↪   proposal id');
        NounsDAOTypes.Proposal storage proposal = ds._proposals[proposalId];

        if (proposal.vetoed) {
            return NounsDAOTypes.ProposalState.Vetoed;
        } else if (proposal.canceled) {
>>          return NounsDAOTypes.ProposalState.Canceled;
```

Tool used

Manual Review

Recommendation

Consider ignoring cancelled proposals, e.g.:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/governance/NounsDAOProposals.sol#L719-L732

```
    function proposalDataForRewards(
        ...
    ) internal view returns (NounsDAOTypes.ProposalForRewards[] memory) {
        require(lastProposalId >= firstProposalId, 'lastProposalId >=
↪   firstProposalId');
        uint256 numProposals = lastProposalId - firstProposalId + 1;
        NounsDAOTypes.ProposalForRewards[] memory data = new
 ↪  NounsDAOTypes.ProposalForRewards[](numProposals);

        NounsDAOTypes.Proposal storage proposal;
        uint256 i;
        for (uint256 pid = firstProposalId; pid <= lastProposalId; ++pid) {
            proposal = ds._proposals[pid];
+           if (proposal.canceled) continue;
```

Rationale is that cancelled proposal is a kind of discarded draft, even if it was voted on, and is to be replaced by another, corrected, version, so including both in the proposal rewards is essentially double counting.

Vetoed ones can stay as is since, apart from veto power holder being trusted, in order to require a veto the proposal needs to be executable, so there looks to be no possibility to quickly iterate many such proposals and dilute the rewards. I.e. canceling provides a significant speed up for eligible proposals making, while vetoing doesn't.

Also, as a simplification and a mitigation for other issues, the eligibility threshold can be passed to `proposalDataForRewards()` and non-eligible proposals can be excluded early on, e.g.:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/client-incentives/Rewards.sol#L323-L327

```
        NounsDAOTypes.ProposalForRewards[] memory proposals =
 ↪  nounsDAO.proposalDataForRewards(
            t.nextProposalIdToReward,
            lastProposalId,
+           $.params.proposalEligibilityQuorumBps,
            votingClientIds
        );
```

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/governance/NounsDAOProposals.sol#L719-L737

```
    function proposalDataForRewards(
        NounsDAOTypes.Storage storage ds,
        uint256 firstProposalId,
        uint256 lastProposalId,
+       uint16 proposalEligibilityQuorumBps,
        uint32[] calldata votingClientIds
    ) internal view returns (NounsDAOTypes.ProposalForRewards[] memory) {
        require(lastProposalId >= firstProposalId, 'lastProposalId >=
 ↪  firstProposalId');
        uint256 numProposals = lastProposalId - firstProposalId + 1;
        NounsDAOTypes.ProposalForRewards[] memory data = new
 ↪  NounsDAOTypes.ProposalForRewards[](numProposals);

        NounsDAOTypes.Proposal storage proposal;
        uint256 i;
        for (uint256 pid = firstProposalId; pid <= lastProposalId; ++pid) {
            proposal = ds._proposals[pid];
+           if (proposal.canceled || proposals.forVotes < (proposals.totalSupply
 ↪  * proposalEligibilityQuorumBps) / 10_000) continue;

            NounsDAOTypes.ClientVoteData[] memory c = new
 ↪  NounsDAOTypes.ClientVoteData[](votingClientIds.length);
            for (uint256 j; j < votingClientIds.length; ++j) {
                c[j] = proposal.voteClients[votingClientIds[j]];
            }
```

Keeping `endBlock > 0` just in case, while the rest is already checked:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/client-incentives/Rewards.sol#L355-L361

```
-           require(block.number > endBlock, 'all proposals must be done with
 ↪  voting');
+           require(endBlock > 0 && block.number > endBlock, 'all voting must be
 ↪  completed');

-           // skip non eligible proposals
-           if (proposals[i].forVotes < (proposals[i].totalSupply *
 ↪  proposalEligibilityQuorumBps_) / 10_000) {
-               delete proposals[i];
-               continue;
-           }
```

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/client-incentives/Rewards.sol#L411-L413

```
        for (uint256 i; i < proposals.length; ++i) {
-           // skip non eligible deleted proposals
-           if (proposals[i].endBlock == 0) continue;
```

Discussion

eladmallel

We are working on a fix that will better filter out proposals.

Still wanted to highlight that we think severity should be lower because likelihood is very low, and that's because an attacker needs to have at least `totalSupply * proposalEligibilityQuorumBps / 10_000` votes.

WangSecurity

Yeam, it seems the likelihood should indeed be low + high impact = medium.

WangSecurity

Comment from LSW:

"the 10% requirement is for coalition that nouns holders can make, not necessary for one attacker. Also, total supply is decreased by forking. Overall, this is a somewhat higher ask, but in the same time the manipulation surface itself is somewhat bigger, as natural proposal count is low enough, so cancelled proposals can have substantial impact (say, 2-3x reward depression for the honest ones)".

Under these reasons, the report remains High severity.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits: https://github.com/nounsDAO/nouns-monorepo/pull/839

dmitriia

Fix looks ok: cancelled proposals are now filtered out in `proposalDataForRewards()`.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-1: Rewards can be allocated for less than minimal reward period with the help of bogus proposal

Source:
https://github.com/sherlock-audit/2024-03-nouns-dao-2-judging/issues/32

Found by

hyh

Summary

Anyone controlling more than `proposalThresholdBPS` share of votes can ignore the minimum reward period to allocate the rewards for the proposal they benefit from.

Vulnerability Detail

The `minimumRewardPeriod` check can be surpassed with the help of any bogus proposal since the check comes before proposal eligibility control. I.e. once current `block.timestamp` is big enough an attacker can just create a proposal only to get past the check by later using that proposal as a anchor for reward distribution call.

This proposal doesn't have to be eligible (can be heavily voted against, can even be vetoed), the only requirement is that it should have ended (`require(block.number > endBlock, ...` check), i.e. the attacker needs to create it beforehand, but the timing is exact and is known in advance, so this can be done all the times. Note that `objectionPeriodEndBlock` is activated only for flipped proposals, which is higly unlikely for bogus ones, so `endBlock = max(proposals[i].endBlock, proposals[i].objectionPeriodEndBlock) = proposals[i].endBlock`, which is known as of time of proposal creation.

Impact

The attack can be carried deterministically, there are no direct prerequisites, while the total cost is additional proposal creation gas cost only (it's not compensated). Surpassing the `minimumRewardPeriod` check can be used for gaming reward allocation, i.e. placing good auctions to the proposal attacker benefit ahead of other proposals, who have to wait for the period to expire.

I.e. in a situation when there are only few active proposals that are about to end, while attacker's one is ended, `$.params.numProposalsEnoughForReward` isn't met, while current auctions have good revenue, the attacker can steal the rewards from other proposals by not waiting for them with the help of the anchor bogus one.

Overall, this setup has medium probability of occurring, while the reward loss can be material, so the impact can be estimated as medium as well.

Likelihood: Medium + Impact: Medium = Severity: Medium.

Code Snippet

`t.lastProposal.creationTimestamp` comes from user supplied `lastProposalId`:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/client-incentives/Rewards.sol#L370-L383

```
        //// Check that distribution is allowed:
        //// 1. At least one eligible proposal.
        //// 2. One of the two conditions must be true:
        //// 2.a. Number of eligible proposals is at least
 ↪  `numProposalsEnoughForReward`.
        //// 2.b. At least `minimumRewardPeriod` seconds have passed since the
 ↪  last update.

        require(t.numEligibleProposals > 0, 'at least one eligible proposal');
        if (t.numEligibleProposals < $.params.numProposalsEnoughForReward) {
            require(
>>              t.lastProposal.creationTimestamp > $.lastProposalRewardsUpdate +
 ↪  $.params.minimumRewardPeriod,
                'not enough time passed'
            );
        }
        $.lastProposalRewardsUpdate = uint40(t.lastProposal.creationTimestamp);
```

With the only checks being `nounsDAO.proposalCount() >= lastProposalId >= t.nextProposalIdToReward`, since `proposalDataForRewards` doesn't filter the proposals, returning all the results sequentially:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/client-incentives/Rewards.sol#L319-L330

```
>>      require(lastProposalId <= nounsDAO.proposalCount(), 'bad
 ↪  lastProposalId');
>>      require(lastProposalId >= t.nextProposalIdToReward, 'bad
 ↪  lastProposalId');
        require(isSortedAndNoDuplicates(votingClientIds), 'must be sorted &
 ↪  unique');
```

```
>>      NounsDAOTypes.ProposalForRewards[] memory proposals =
 ↪  nounsDAO.proposalDataForRewards(
            t.nextProposalIdToReward,
            lastProposalId,
            votingClientIds
        );
        $.nextProposalIdToReward = lastProposalId + 1;

>>      t.lastProposal = proposals[proposals.length - 1];
```

User that can create proposals can also anyhow time the bogus one:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/governance/NounsDAOProposals.sol#L871-L881

```
    function createNewProposal(
        ...
    ) internal returns (NounsDAOTypes.Proposal storage newProposal) {
        uint64 updatePeriodEndBlock = SafeCast.toUint64(block.number +
 ↪  ds.proposalUpdatablePeriodInBlocks);
        uint256 startBlock = updatePeriodEndBlock + ds.votingDelay;
>>      uint256 endBlock = startBlock + ds.votingPeriod;
```

Tool used

Manual Review

Recommendation

Consider basing the check on the last eligible proposal, not just the user supplied one, e.g. by filtering them out in `proposalDataForRewards()`, which isn't used outside reward logic:

https://github.com/sherlock-audit/2024-03-nouns-dao-2/blob/main/nouns-monorepo/packages/nouns-contracts/contracts/governance/NounsDAOProposals.sol#L719-L737

```
    function proposalDataForRewards(
        NounsDAOTypes.Storage storage ds,
        uint256 firstProposalId,
        uint256 lastProposalId,
+       uint16 proposalEligibilityQuorumBps,
        uint32[] calldata votingClientIds
    ) internal view returns (NounsDAOTypes.ProposalForRewards[] memory) {
        require(lastProposalId >= firstProposalId, 'lastProposalId >=
↪   firstProposalId');
        uint256 numProposals = lastProposalId - firstProposalId + 1;
        NounsDAOTypes.ProposalForRewards[] memory data = new
↪   NounsDAOTypes.ProposalForRewards[](numProposals);

        NounsDAOTypes.Proposal storage proposal;
        uint256 i;
        for (uint256 pid = firstProposalId; pid <= lastProposalId; ++pid) {
            proposal = ds._proposals[pid];
+           if (proposal.canceled || proposals.forVotes < (proposals.totalSupply
↪   * proposalEligibilityQuorumBps) / 10_000) continue;

            NounsDAOTypes.ClientVoteData[] memory c = new
↪   NounsDAOTypes.ClientVoteData[](votingClientIds.length);
            for (uint256 j; j < votingClientIds.length; ++j) {
                c[j] = proposal.voteClients[votingClientIds[j]];
            }
```

In order to minimize the changes this suggestion is shared with another issues.

Discussion

eladmallel

Thanks for for bringing this up! This issue's fix will probably be the same as #46, with better proposal filtering.

dmitriia

This and #46 have the same fix, which is included in the #51 recommendation (3rd code snippet).

I.e. for #32, #46 and #51 set please use the fix from #51 as it covers them all.

WangSecurity

@dmitriia want to confirm that these three issues are not dups, even though they have the same fix?

dmitriia

@WangSecurity They are different: 32 is for the avoidance of minimal reward period, 46 is for extending auction revenue span, while 51 is for manipulation with the help of proposal canceling.

They can have different fixes, but the recommendation is just unified to minimize code changes. Although, the base recommendation for 51 by itself doesn't include the 32 and 46 fixes, but there is an expanded write up of the cumulative change put there, that includes some logic optimization along with these fixes.

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/nounsDAO/nouns-monorepo/pull/839

dmitriia

Fix looks ok.

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.